# A comparison of AES candidates on the Alpha 21264

Richard Weiss
VSSAD Labs
Compaq Computer Corp,
334 South St
Shrewsbury, MA 01545
Richard.Weiss@Compaq.com

Nathan Binkert
Computer Science Dept
University of Michigan
Ann Arbor, MI
binkertn@umich.edu

## ABSTRACT

We compare the five candidates for the Advanced Encryption Standard based on their performance on the Alpha 21264, a 64-bit superscalar processor. There are several new features of the 21264 that have a significant impact on encryption/decryption speed. The main ones are greater potential for instruction-level parallelism (ILP) and larger level 1 cache. The ILP comes from the fact that the 21264 can issue four integer instructions per cycle. We envision that for high-performance servers, there will be multiple streams of data for encryption or decryption. The type of parallelism that we consider in this paper is the encryption of multiple, independent blocks interleaved in the same code loop running on the *same processor*. This benefits some algorithms more than others. Rijndael and Twofish turn out to be the fastest for a single block at a time, but RC6 is potentially the fastest when processing two blocks at a time. The reason for this is that out-of-order execution together with an issue width of four can be used to hide the latency of integer multiplies.

## Introduction

The new AES algorithms will be used on a wide range of CPU's. The Alpha 21264 is a good representative of a 64-bit RISC architecture. Its features include a 64K two-way set associative level-1 cache, the capability to issue 4 integer instructions each cycle, and out-of-order execution. Since the Alpha is most likely to be used in servers, it will probably be used for encrypting or decrypting multiple streams of data simultaneously. This can be done on multiple processors, but it is also relevant to look at the efficiency of processing more than one block simultaneously on each processor, thus increasing the throughput of the system. In the remainder of this paper, we will use the term ***multiple stream*** or ***multistream*** to refer to more than one block on the same processor. Most of the studies so far have looked at single stream performance, where latency is the dominant factor. In order to get optimal multistream performance, it will be necessary to harness the full bandwidth of the processor. The five candidate AES algorithms have different computational requirements, and therefore have different behavior with respect to multistream than single stream.

We illustrate the multiple stream scenario with an example, so that there is no ambiguity. Consider the following assembly language fragment from a loop for an imaginary processor that can issue two instructions per cycle, at most one of which can be a multiply:

```
loop:
      1. Load S[0]         # load key
      2. T = Mull A*A

      3. Load S[1]         # load key
      4. U = Mull B*B

      5. C = Shift_right  T
      6. D = Shift_left   T

      7. E = Shift_right  U
      8. F = Shift_left   U

      9. C = C Or D
      10. E = E Or F

      11. B = C Add S[0]
      12. A = E Add S[1]

      13. Br loop
```

The processor will execute two instructions per cycle except for the branch. If the latency of each instruction were one cycle, then the whole code would take seven cycles. However, if the latency of a multiply is seven cycles and at most one can be issued in a given cycle, then there is a five cycle stall after the fourth instruction. Therefore, the execution time increases to 12. Now consider what we can do for two independent blocks of data:

```
loop:
      Load S1[0]         # load key1
      T1 = Mull A1*A1

      Load S1[1]         # load key1
      U1 = Mull B1*B1

                        C2 = Shift_right  T2
                        D2 = Shift_left   T2

                        E2 = Shift_right  U2
                        F2 = Shift_left   U2

                        C2 = C2 Or D2
                        E2 = E2 Or F2

                        B2 = C2 Add S2[0]
                        A2 = E2 Add S2[1]

                        Load S2[0]         # load key2
                        T2 = Mull A2*A2

                        Load S2[1]         # load key2
                        U2 = Mull B2*B2

      C1 = Shift_right  T1
      D1 = Shift_left  T1

      E1 = Shift_right  U1
      F1 = Shift_left   U1

      C1 = C1 Or D1
      E1 = E1 Or F1

      B1 = C1 Add S1[0]
      A1 = E1 Add S1[1]

      Br loop
```

The combined loop can process two blocks in only 13 cycles. The processing of the two blocks can be overlapped in such a way that while the shift operations for one block are waiting for the multiplies to complete, operations on the other block can proceed. For the 21264, the latency for a multiply is actually seven, and the latency of a load is three or more, depending on whether or not the value is in the D-cache. The 21264 can issue up to four integer instructions in one cycle, at most two of which can be loads. The out-of-order processing capability is not actually used if the compiler schedules the instructions to take into account the latency. It should be noted that future generations of Alpha processors will have simultaneous multithreading (SMT), which will eliminate the necessity of the programmer/compiler merging two streams of data in one instruction stream.

The key to taking advantage of the full issue width of the Alpha is recognizing when a program has a low number of instructions per cycle (ipc). In the above example, this was caused by the long latency of the multiplies, but there may be other cases where this happens. For example, in the implementation of Serpent that we used, there were long chains of dependent logical operations, which resulted in an ipc of slightly less than two. Thus, Serpent can achieve a speedup of almost two by processing two streams. RC6 is similar to the example above in that the multiplies introduce latency, which reduces the ipc to a level for which processing two streams works well. On the other hand, Rijndael, Twofish and Mars do not lend themselves to this approach. They can be coded efficiently for single stream so that the table lookups can be overlapped with the other computation and the ipc is well over two. It should be noted that an ipc of greater than two does not preclude multistream processing, but the gains are likely to be small. Also, it is important to use an optimized version of the code, otherwise a low ipc will only reflect the inefficiency of the implementation rather than the potential for multistream parallelism. For this reason, we examine assembly language implementations in addition to the C versions.

One of the architectural features that is missing from Alpha is the 32-bit rotate. This requires several instructions to emulate. A fixed rotation requires two shifts an “and” and an “or”. These can be executed in two parallel chains and in the absence of other parallelism they have an ipc of two.

The next section presents an analysis of each algorithm in terms of ipc for a C implementation and for an assembly code implementation.

## Analysis of Algorithms

Our goal is to get a quick estimate of the performance for multistream data. We do this by checking the timings for the Gladman C implementations of the five candidate algorithms for single stream data and estimating the ipc. Then in some cases, we also look at assembly language implementations to see if the ipc could be increased.  While a high ipc will rule out a gain from multistream, a low ipc does not guarantee one.  A range of techniques was used from a complete implementation in assembly language in the case of Rijndael, to coding a single round in assembly language for Rc6 and Twofish, to a data dependency anlysis for Mars and Serpent.  The data dependency analysis together with instruction latency was used to estimate optimal times for the last two algorithms.  In the one case where we did an assembly language implementation, the time for this was compared with our estimate.  Finally, we estimated the gains for multiple stream implementations.

**Mars**
The Mars algorithm has three phases: simple arithmetic and logical operations, table lookup and rotations.  The table lookup, which is mixed with some fixed rotations has a four-fold parallelism.  This seems to be the reason for a high ipc, and therefore little gain from multistream.  Since the Alpha does not have a 32-bit rotate, this increases the number of instructions.  For this reason, it is both one of the fastest algorithms on a Pentium Pro but one of the slowest on the 21264.

**RC6**
RC6 turns out to be a lot more efficient on the Alpha 21264 than expected from observing the number of cycles for a single block of data. For single stream performance, each round when coded in assembly language, takes 18 cycles and there are 20 rounds.  If we allow 20 cycles for setup, this gives a total of 380 cycles per block.  This is amazingly close to the current reported figure of 382 cycles per block for the optimized C version.  A single round of encryption for two independent blocks of data simultaneously was also coded in assembly language for an estimated 21 cycles, which is less than 11 cycles/block. For 20 rounds, this would be 210 cycles/block plus the time for setup and storing results.  This is as fast as Rijndael, and is potentially more consistent since it uses multiplication, which have a fixed latency, and does not depend on table lookups which could suffer occasional cache misses.  In addition, if the algorithm were used with a word size of 64, this could potentially double the throughput, since the 64-bit versions of the operations multiply, xor, add and rotate are as fast or faster than the 32-bit versions on Alpha processors.

**Rijndael**
The simplicity of the Rijndael algorithm makes it easy to analyze.  We were able to produce an efficient implementation in assembly code together with timing results.  The major computational cost for this algorithm is accessing the look-up tables.  This can be done in three instructions: extract byte, add to base address, and load the value. For Alpha, this is relatively fast, since the tables fit in the level-one cache.  Ideally, one round of Rijndael could be done in 18 cycles: however, in practice, this requires tuning the code to eliminate I-cache misses, D-cache misses, etc.  What we observed was that the code took 246 cycles/block when executed repeatedly.  This is about 23 cycles per round. This was the fastest algorithm we have observed for 128-bit key length. However, since the number of rounds for Rijndael depends on the key length, this is not the fastest for all applications.

We expect the Rijndael algorithm to scale well with future processors since the makeup of the code is such that one quarter of the instructions are loads. The Alpha 21264 can issue four integer instructions per cycle, and there is a four-fold parallelism from the four S-boxes.  However, this gives it a high ipc and means that there is little gain from multistreaming.  A single round of

Rijndael takes 18 cycles.  The setup and exit code adds another 30 cycles to the total to give approximately 210 cycles per block.

**Serpent**
Based on the C-code from Brian Gladman, this algorithm is the slowest.  However, it speeds up very well with multistreaming.  The S-boxes are implemented by sequences of bit-parallel logical operations.  Due to data dependencies in this code, the ipc is slightly less than two.  The technique for estimating the two stream performance was to modify the C code.  Each round is composed of three macros: an “xor” with the key, an S-box computation, and a linear transform. The processing of the two streams was interleaved by repeating each macro for the first stream with the identical macro for the second stream.  The compiler was able further mix the instructions to eliminate stalls.  Nevertheless, Serpent remains one of the slower algorithms because of the large number of rounds and the large number of instructions per round.  It should be noted that most of the operations in Serpent operate on bits in parallel.  It should be possible to process two blocks of 32-bit words by using the full 64-bit data path.  Namely, one block would use the upper 32 bits, and the other block would use the lower bits.  There would be an extra “and” for the rotates as well as packing the two words together, but the speedup could be close to 2x.

**Twofish**
Based on an assembly language coding of a single round, twofish performs approximately as well as Rijndael on both the 21164 and the 21264 for 128-bit key length.  Since Twofish does not require more rounds for larger key lengths, its relative performance would be better for longer keys. It can potentially do eight S-box lookups in parallel for each round.  This gives it a high ipc and small gain for multistreaming.

## Timing Results

Table 1 shows the results from optimized C-code for the Alpha 21164 and 21264 processing one block at a time.  The 21164 can issue two integer instructions per cycle and the 21264 can issue four.  The results are similar to those published by Granboulan [Gran]. Our timings were all obtained by running each of the algorithms for key setup, encryption and decryption on a single stream of data, one block at a time.  The C-versions of these algorithms are the ones published by Gladman [Glad1].  We ported them to Alpha by using the native cycle count register and modifying the declarations to eliminate alignment errors in the code.  The basic idea is to time the execution of the encryption (decryption) code running once, then time it running twice.  The minimum times over a large number of iterations are subtracted to measure the time to execute the code without the startup costs.  In addition, the encryption (decryption) code is run once at the beginning to warm up the caches.

In order to relate our assembly code estimates to the C implementations, we linked our assembly version of Rijndael to the Gladman harness and observed an encryption time of 280 cycles/block.  The assembly code when executed for a large number of iterations took a minimum of 246 cycles/block.  This suggests that the C++ overhead for calling some of the C or assembly functions could be significant.

In Table 2, we have estimated timing results for assembly language implementations for some of the algorithms for single stream.  Table 3 shows the estimated timing for assembly code for processing multiple streams.

| EV56 (21164) | Mars | RC6 | Rijndael | Serpent | Twofish |
|---|---|---|---|---|---|
| Ours | 701c | 571c | 439c | 984c | 442c |
| Granboulan website | 507c | 559c | 490c | 998c | 490c |

| EV6 (21264) | Mars | RC6 | Rijndael | Serpent | Twofish |
|---|---|---|---|---|---|
| Ours | 515c | 428c | 293c | 854c | 316c |
| Granboulan website | 450c | 382c | 285c | 855c | 315c |

Table 1. Timing comparison in cycles/block for C code.

| EV6 (21264) | Mars | RC6 | Rijndael | Serpent | Twofish |
|---|---|---|---|---|---|
| Assembly code | 375c | 360c | 210c | 570c | 255c |

Table 2. Estimated timing for assembly code in cycles/block.

| EV6 (21264) | Mars | RC6 | Rijndael | Serpent | Twofish |
|---|---|---|---|---|---|
| Assembly code | 375c | 210c | 210c | 506c | 255c |

Table 3. Estimated time for assembly code encrypting two blocks simultaneously. Times are in cycles/block.

## Conclusions

RC6 has the most potential for parallelism when multiple streams are processed on the same processor simultaneously in a single thread. One reason for this is that it relies heavily on multiplication, which itself has a large degree of parallelism for the Alpha processors. 32-bit multiplies are inherently parallel because they operate on four bytes at the same time. Using 64-bit multiplication would afford even more parallelism. The 21264 can issue one multiply every cycle. The latency of seven cycles does not limit bandwidth for this algorithm in multistream mode. An S-box lookup requires three instructions, and only operates on one byte at a time. Note that while RC6 has variable 32-bit rotations, one of the intermediate results from the fixed rotation by 5 is re-used in the variable rotation.

Serpent also has a large gain from multistream processing because of the long dependent chains of instructions and low ipc. However, because of the large number of rounds and instructions per round, it still is slow.

Following RC6 are Twofish and Rijndael, which both use 8-bit table lookups and linear transforms. Twofish has an advantage for longer keys, but Rijndael seems the fastest for 128-bit keys. Based on an assembly language implementation of

Rijndael, there can be a significant difference between the estimated performance and what can be readily achieved/observed by counting cycles outside of the algorithm function call. Comparing code execution with timing estimations can have a significant amount of error.

Since our estimates for the Alpha 21264 are based on instruction level parallelism for processing multiple streams, similar behavior should be observable for Itanium and other VLIW machines.

**Acknowledgements.**
We would like to thank Dr. Brian Gladman for publishing unified C implementations of the five AES candidate algorithms. Also we thank Steve Root for assembly language implementations of some of the algorithms.

## References

[KA] Almquist, Kenneth. "AES Candidate performance on the Alpha 21164.
http://home.cyber.ee/helger/aes/kenneth.txt

[Glad1] Gladman, Brian. "Implementation experience with AES candidate algorithms." Second AES Conference, Feb, 1999.
http://jya.com/bg/gladman.pdf

[Glad2] Gladman, Brian.
http://www.btinternet.com/~brian.gladman/cryptography_technology/Aes/index.htm

[Gran] Granboulan, Louis. "AES Timings of best known implementations."
http://www.dmi.ens.fr/~granboul/recherche/AES/timings.html

[SKW] Schneier, B., Kelsey, J., Whiting, D., et al. "Performance Comparison of the AES Submissions."