Fast Implementations of AES Candidates

Kazumaro Aoki[1] and Helger Lipmaa[2]

[1] NTT Laboratories
1-1 Hikarinooka, Yokosuka-shi, Kanagawa-ken, 239-0847 Japan
maro@isl.ntt.co.jp
[2] Küberneetika AS
Akadeemia tee 21, 12618 Tallinn, Estonia
helger@cyber.ee

Abstract. Of the five AES finalists four—MARS, RC6, Rijndael, Twofish—have not only (expected) good security but also exceptional performance on the PC platforms, especially on those featuring the Pentium Pro, the NIST AES analysis platform. In the current paper we present new performance numbers of the mentioned four ciphers resulting from our carefully optimized assembly-language implementations on the Pentium II, the successor of the Pentium Pro. All our implementations follow well-defined API and timing conventions and sensible guidelines, like no using of self-modifying code and key-specific static data — i.e., tricks that speed up the implementation but at the same time restrict the field of application. Our implementations are up to 26% percent faster than previous implementations. Our work also shows how a simple change (inclusion of the MMX technology) in the analysis platform can influence the relative encryption speed of different ciphers. To enable everyone to compare their implementations to ours, we also fully specify our procedures used to obtain the speed numbers.

1 Introduction

For more than 20 years, DES [FIP77] has been a widely employed cryptographic standard. While the best cryptanalytic attacks against DES (differential and linear cryptanalysis) are still highly impractical, during the last years DES has became obsolete for its too short key and block sizes, not withstanding the current advances in computing technology. Motivated by this, NIST initiated a new effort to replace DES as a standard. 21 algorithms were submitted and 15 algorithms were accepted as AES (*Advanced Encryption Standard*) candidates, of which 5 candidates—MARS [BCD+98], RC6 [RRSY98], Rijndael [DR98], Serpent [ABK98], Twofish [SKW+99b]—were chosen to the second round.

However, the AES process was started not only due to the theoretical reasons: there are a few well-known constructions, including 3DES, that seem to have very good security margins. Unfortunately, 3DES, based on the hardware-oriented DES, is unsatisfyingly slow on the modern 32- and 64-bit computer architectures: modern block ciphers are up to 10 times faster than 3DES. Regardless of these ciphers having unproven (even by time) security properties, they are widely used in the industry by pragmatic reasons: hardware applications like 1 GBits/s Ethernet or on-the-fly encryption of 160 MByte/s

SCSI hard disks are requesting for faster ciphers. Clearly, the situation of having a (moderately) secure and (moderately) fast *de jure* standard DES, a (probably) secure and (clearly) slow *de facto* standard 3DES and some fast but with unknown security margin *de facto* standards is not acceptable: there should be a single standard that is both secure and fast. This is one of the reasons why, when inviting the public to propose candidates for the AES, NIST explicitly stated that the new standard should be both "more secure and faster" than 3DES.

While security of the candidates cannot be exactly quantified by the currently known methods, it seems to be easier to measure their speed. However, there is still a lot of ambiguity in answering the question what AES candidate is the fastest. Several papers (including [Lip99,SKW+99a]) have compared AES candidates speed, but since the implementations quoted in them are often incomparable (or based on pure estimations), one cannot make direct conclusions about the efficiency of the ciphers based on the published papers. Incomparability stems from the different implementation assumptions, API's, hardware (e.g., processors) and software (e.g., compilers) used by implementers. Even more, some of the timings presented in previous papers correspond to "show-case" (as opposed to practically applicable) implementations, some examples of those being the fastest implementation of Twofish [SKW+99b] that uses self-modifying code and Brian Gladman's implementations of AES candidates [Gla99] that use a number of key-specific static variables instead of allocating a register to address them, therefore effectively freeing some registers for other uses. Especially in the case of the Pentium family, where the number of available registers is very restricted, such implementations may result in a huge speed up. However, both types of implementation tricks restrict the application area of the implementation.

In the current paper we try to give a satisfactory answer to the question "what cipher is the fastest on the Pentium II" by carefully optimizing the 4 fastest AES candidates—MARS, RC6, Rijndael and Twofish—in Pentium II assembly, using for all implementations exactly the same, reasonable in practice, API and speed measurement conditions for all the ciphers. Due to this, our results are much fairer than most of the previously known ones: our implementations can be seen as black boxes applicable in almost any possible application of block ciphers on an environment featuring Pentium II. Additionally, careful optimization process resulted in implementations that are clearly faster than the previously known implementations. (Except for Twofish, which has still a faster "show-case" implementation.)

We start the paper by describing our platform of choice (Section 2), implementation philosophy and API (Section 3). Section 4 briefly surveys our results, and Section 5 gives more details on the problems encountered when implementing the ciphers. More information about the Pentium II is given in the Appendices.

2 Choice of the Platform

Our first principal choice was the decision what processor to use. By purely pragmatic reasons we decided that the implementation environment equips an Intel Pentium family CPU: while this family is not the most modern processor family available, it is the most widespread one at the moment of writing this paper and most probably also during the

next few years. Therefore, since in the foreseeable future most of the software-based commercial security applications run on the Pentium family (as recognized also by the AES finalists designers), this family has the most direct impact on the choice of a cipher by security consumers.

At second, from the Pentium family we decided to choose the Pentium II processor. At first, it is a more advanced processor than Pentium Pro, the NIST AES analysis platform: the Pentium II provides (twice) larger register space due to the added MMX technology, and many new MMX-specific commands. Compared to the Pentium Pro, the Pentium II is also easier to obtain at the current stage, since Pentium Pro has been out of the manufacturing for a while. On the other hand, the Pentium II was preferred by the authors to the Pentium III since the latter is somewhat too new and controversial due to the privacy issues.

Another reason to choose Pentium II was that as the successor of the NIST AES analysis platform, implementing the AES candidates on the Pentium II could provide some insights on how generally suitable are the candidates, some of which were specifically optimized for the Pentium Pro, on future processors having features unpredicted by algorithm designers. While this is not as crucial as withstanding the "future attacks", it still gives some ideas about the possible longevity of the cipher. (We clearly would not want the AES in 20 years to have the role the 3DES has today!)

As shown in [Lip98], the MMX technology can seriously speed up IDEA ([LM90], [LMM94]), one of the believably most secure block ciphers with 64-bit block size. As stated in [Lip98], this can be done since IDEA has its key attributes similar to those of multimedia applications, for which the MMX technology was originally created. An open question posed in [Lip98] was how much would the MMX technology help implementing other ciphers, including the AES candidates. In the following we will partially answer to that question, showing that also some ciphers using only "simple" operations can greatly benefit from the added MMX technology. A short overview of Pentium II that is necessary for implementers and for cryptographers who design ciphers optimized for this platform is given in Appendix A. We refer for Intel manuals for a more complete overview.

3 Implementation Considerations

Several papers (including, in particular, [Lip99,SKW+99a]) have compared AES candidates speed, but since the implementations quoted in them are often incomparable (or based on pure estimations), one cannot make direct conclusions about the efficiency of these algorithms based on the published papers. Incomparability stems from the different implementation assumptions, API's, hardware (processors) and software (compilers) platforms used by implementers. Even more, some of the numbers there correspond to the "show-case" (as opposed to practically applicable) implementations; including the bizarre case that one candidate was claimed to be the fastest on its inventors laptop under some suitable conditions.

As another example of the unsuitability of some "show-case" implementations, the fastest implementation of Twofish [SKW+99b] uses self-modifying code and therefore cannot be used in a number of applications, while Brian Gladman's implementations of

AES candidates [Gla99] use a number of key-specific static variables instead of allocating a register to address them, therefore effectively freeing some registers for other uses. Especially in the case of the Pentium family, where the number of available registers is very restricted, such implementations may result in a huge speed up. On the other hand, Gladman's implementations cannot be used several applications, including multithreaded programs and SMP (symmetric multi-processing) systems.

Most of the security customers need however speed numbers applicable in whatever product they use in whatever environment in runs (for example, in a Linux kernel-supported IPSEC implementation, secure login or multithreaded access to encrypted storage arrays). For users it is necessary to know in what environment the measured speed numbers were obtained, to be able to calculate the possible efficiency of the ciphers in their own environments. Additionally, full specification is important for other implementers to be able to compare their implementations with ours. Hence, apart from providing "clean" implementations under some reasonable public assumptions, we shall also next fully specify these assumptions:

- We do not use self-modifying code ("code compilation" [SKW+99b]) since it makes the implementation inapplicable in a number of situations, e.g., in operation-system kernel and ROM-based applications.
- We additionally decided not to use key-specific static areas since then the implementation could not be used, e.g., in SMP-capable systems and multithreaded programs.
- We decided to maximally use the MMX technology since it should not be forbidden in any reasonable modern environment. (While using self-modifying code and key-specific static areas is generally considered to be a bad programming practice.)
- We decided to use exactly the same API (specified later in Section 3.1) in all our implementations.
- A number of well-understood assumptions that 1) improve the speed and can be easily followed by implementers or 2) are essential to even be able to measure the speed:
  - All codes and data are correctly aligned.
  - Input and output texts and codes are preloaded to L1 cache in the possible extent to reduce the number of cache misses.
  - Simplicity of code: we tried to reduce time spent during writing and optimizing the code. In particular, all our implementations use highly optimized but round-number independent round macros. (Hence, our results could be slightly bettered if every round would optimized separately to avoid, e.g., delays in fetching stage.)

3.1 API

Since a different API can be influence the speed of an implementation severely, we also decided to fully specify the API used by us to make for the other implementers easier to compare their implementations to the ours. We felt that this is necessary, since AES candidate implementations reported in [Lip99] vary greatly in their API's.

```
void xxKS(char *master, uint32 bitLen, char *eKey);
void xxEnc(char *inBlk, uint32 lenBlk, char *eKey,
           char *outBlk);
void xxDec(char *inBlk, uint32 lenBlk, char *eKey,
           char *outBlk);
```

where

xx is algorithm name (e.g., `Rijndael`).
xxKS is key scheduling subroutine.
xxEnc is encryption subroutine that encrypts `lenBlk` blocks of plaintext starting from the address `inBlk` to the ciphertext location `outBlk`, by using extended key `eKey`, in ECB block cipher mode.
xxDec is decryption subroutine with the same input conventions as `xxEnc`.
uint32 is the type of 32-bit unsigned integers (in the case of Pentium II, equal to `unsigned long` in the case of most compilers).
master is pointer to the master key bits.
bitLen is the bit length of a master key.
eKey is pointer to subkeys and other initialization data, used later by encryption and decryption.
inBlk is pointer to input texts to be encrypted in the case of `xxEnc` and to be decrypted in the case of `xxDec`.
outBlk is pointer to the corresponding output texts.
lenBlk is number of blocks to be encrypted or decrypted.

Fig. 1. Specification of our API.

Note that our API, depicted in Figure 1, is essentially equivalent to the API's used in most of the commercial applications, specifying only those inputs and outputs to the algorithms that are really needed by the algorithms. (Names of the subroutines and their parameters of course do not affect the speed, of course.) Our API was fixed for the key length of 128-bits due to the feeling that at the time when greater key sizes become necessary, our implementation platform would already be a history.

Here, the key schedule and decryption subroutines are specified only for completeness. Since in the current paper we are not interested in the optimization of these subroutines, we almost do not mention decryption and key schedules hereafter.

3.2 How to Measure a Number of Cycles

Different time measurement methods may change the speed numbers quite dramatically. As in the case of the API's, we decided to use one, sensible published and *fully specified* convention (specified in Figure 2) for all the implementations. (Note that this wrapping corresponds almost exactly to the method specified in [Fog00], to which the reader is referred for a throughout explanation of the method.) The inputs and key of the cipher are generated randomly before the measurement begins, to prevent "optimization" for specific class of keys. The input variable `lenBlk` was chosen to be equal to 8000 so that the input and output texts would not fit in the L1 cache. Also, `time` is a work area of type `uint32`, used in later calculations.

```
movd mm0, dword ptr [time];  /* warm cache and set MMX state */
xor eax, eax;
cpuid;                       /* serialize instructions */
rdtsc;                       /* read time-stamp counter */
mov dword ptr [time], eax;   /* save counter */
xor eax, eax;
cpuid;                       /* serialize instructions */
/* xxEnc() or xxDec() */
xor eax, eax;
cpuid;                       /* serialize instructions */
rdtsc;                       /* read time-stamp counter */
sub dword ptr [time], eax;   /* compute the difference */
emms;                        /* empty MMX state */
```

Note that `time` is a 4 bytes work area.

Fig. 2. Time measurement code

```
   /* push all used registers */
   cmp dword ptr [lenBlk], 0;
   jz L1;
   align 16;
L0:
   dec dword ptr [lenBlk];
   jnz L0;
L1:
   /* pop these registers once more */
```

Fig. 3. Null function

Note that this method has some overhead, due to both high latency of the `rdtsc` instructions and also the overhead caused by looping instructions like `jnz` which are not formally part of the cipher itself. (Looping instructions can be seen as a part of the block cipher mode, however.) We measure this overhead by using the null function shown in Fig. 3 obtaining a value `nulltime`, and then we subtract it from the value of `time` obtained by measuring the speeds of different encryption/decryption procedures. Finally, this result is divided by the number of blocks encrypted. Intuitively, by using this method we obtain the number of cycles corresponding to unrolled implementation of the block cipher, or to the implementation where we only care about the time encrypting one block takes without adding any extra overhead. (Note that the subtracted overhead number was equal to ≈ 6 in the case $n = 8000$. One could easily add this number to those presented later to get the number of cycles *with* overhead.)

Chosen time measurement method is also reasonable in practice: when the value of `lenBlk` was chosen to be different, for most of the implementations (*including* the implementation of null cipher), the execution times increased by almost the same constant. Hence, the null cipher proved experimentally to be well-defined.

Cipher	Mbits/s on a 450 MHz Pentium II	Cycles per block	Best previous result	Speedup
Null cipher	—	6	—	—
RC6	258 Mbits/s	223	243 [Riv98]	8%
Rijndael	243 Mbits/s	237	320 [DR98]	26%
Twofish	204 Mbits/s	282	315 [SKW$^+$99b]	11%
MARS	188 Mbits/s	306	390 [BCD$^+$98]	22%

Table 1. Performance in clock cycles per block of output of four AES finalists. (Only encryption considered)

Finally, we did a loop of 500 times over the described measurements and then chose the smallest number for every cipher, since that corresponds most likely to the case where most of the data and code are in L1 cache and the branch prediction works successfully: i.e., to the bulk encryption speed of the cipher itself.

4 Implementation Results

From the five AES finalists, one (Serpent) is regarded as a very conservative design but at the same time also being clearly slower than the other AES finalists. Rest of the finalists have comparable timings on most of the modern computer platforms, where one of the ciphers is the fastest in one platform, and another one in another platform. Since also on the Pentium II processor, Serpent seems to be very slow by the published data, we decided postpone its implementation to the future and concentrate on the fast ciphers.

Timings, obtained by measuring the speed of implementations by following previously specified procedures are summarized in Table 1[1]. The numbers in the middle columns show how many cycles it takes to encrypt one 128-bit block by using the chosen cipher with a 128-bit key. These results indicate that the chosen four AES finalists are extremely fast. For comparison, the standard hash algorithm SHA-1 *hashes* a 512-bit block in 837 cycles (i.e., 13.1 cycles per byte) and DES and 3DES encrypt a 64-bit block respectively in 340 and 928 cycles (resp., 42.5 and 116 cycles per byte) [PRB98], while RC6 and Rijndael respectively encrypt a 128-bit block in 223 and 237 cycles (resp., 13.9 and 14.8 cycles per byte). However, note that the cited timings in [PRB98] were obtained on a plain Pentium and therefore could most probably be improved on the Pentium II.

Our results seem to indicate, that the speed difference between different ciphers is less than expected: as before, RC6 is still the fastest cipher on the Pentium II, but the difference between it and Rijndael has decreased seriously. Hence we hesitate to say that RC6 is the fastest cipher. However, based on the cited results, we can classify the ciphers to two groups: blastingly fast ciphers RC6 and Rijndael and somewhat slower, but still very fast ciphers Twofish and MARS.

[1] We also started to code the decryption routines, finishing RC6 decryption (209 cycles per block) and Twofish decryption (276 cycles per block).

However, one has to keep in mind that RC6 and MARS have design features that make them specifically efficient on the Pentium Pro (and its successors), while their performance seriously degrades on other processors [Lip99,SKW+99a]. This is due to the use of complex instructions (32-bit multiplication and data-dependent rotation) that are cheap on the P6 family (Pentium Pro, Pentium II, Celeron, Xeon and Pentium III) but very expensive on most of the other platforms. Interestingly, also the next generation Pentium processor (code-named "Willamette", [Int00]) has latency 10 multiplication and latency 2 or 4 shifts, as compared to latency 4 multiplication and latency 1 shifts on the P6 family [Int00, Section 4.1.3]. Hence, RC6 and MARS would considerably slow down on the Willamette, the next generation Pentium family processor. On the other hand, Rijndael and Twofish are based on simple operations, and run equally well on all platforms. The speed ratio between Rijndael and Twofish seems be remain *almost* the same on the other platforms [Lip99] (namely, Rijndael being $5 \ldots 25\%$ faster than Twofish).

Note that the speed up percents in Table 1 correspond to the achieved speed ups compared to the fastest "clean" implementations (i.e., those not using key-specific static data or self-modifying code). However, these percents do not always mean that our implementation techniques were exactly as much better. For example, the best previous implementation of Rijndael was done for the plain Pentium, but not for the Pentium Pro: a factor that may have negatively affected its performance. The best previous "clean" implementation of MARS was written in C, and therefore had also a relatively slow performance. However, our own C implementation of MARS is clearly faster than the one given in Table 1. In the case of Rijndael, most of the acceleration Rijndael is due to the efficient use MMX technology. In general, speed up comes mainly from better optimization (elaborated tradeoff between processor operating stages) and full usage of the Pentium II possibilities (i.e., the MMX technology).

To further clarify how does the Pentium II architecture impact the speed, Table 2 shows the detailed information of our implementations in encryption mode in the micro-operation level. Usage of the table is simple. For example, in the intersection point of "@round" row and "port 01" column in `TwofishEnc` table one would find 19. That means that there are 19 μoperations in the round function of `TwofishEnc` which will be executed on port 0 or port 1.

Interestingly, our implementations of MARS, Rijndael and Twofish all require approximately the same number of μoperations, while RC6 is about two times "better" in this category. On the other hand, RC6 is also the worst cipher to parallelize: while in Rijndael, more than 2.5 μoperations are executed per a cycle, RC6 can only mildly use the super-scalar parallelism of Pentium II. More cipher-specific comments will be given in the next.

5 Cipher-Specific Comments

5.1 MARS

In the case of MARS [BCD+98], the speed difference between a carefully optimized C implementation (using a recent snapshot of the `gcc` compiler) and an optimized assembly language implementation is only about 11% on the Pentium II. The speedup

	port 0	port 1	port 01	port 2	port 3	port 4	total
MARS encryption							(1.87 μops/cycle)
prewhitening			5	8			13
forward mixing	16		77	32			125
@core ($\times 16$)	6		9	3			18
backward mixing	16		85	32			125
postwhitening		1	8	4	4	4	21
total	128	1	319	124	4	4	572
RC6 encryption							(1.47 μops/cycle)
prewhitening			2	7			9
@round ($\times 20$)	8		5	2			15
postwhitening		1	4	5	5	5	20
total	160	1	106	52	5	5	329
Rijndael encryption							(2.54 μops/cycle)
whitening		1	8	6			15
@round ($\times 9$)	4	1	34	19			58
last round	4	3	31	20	3	3	64
total	40	13	345	197	3	3	601
Twofish encryption							(2.11 μops/cycle)
prewhitening			5	8			13
first round	5		19	10			34
@round ($\times 15$)	6		19	10			35
postwhitening	2	1	8	4	4	4	23
total	97	1	317	172	4	4	595

Table 2. Number of μoperations in our implementations

comes mainly from a slightly more efficient allocation of the integer registers and some (minimal) usage of the MMX instructions in the assembly implementation. However, the MMX technology is only moderately useful for MARS, since the complex instructions performed in MARS (i.e., 32-bit multiplication, data-dependent rotation and S-box lookups) are not available for the MMX registers. Additionally, due to the high data-dependency there is very limited freedom in meaningfully rescheduling the instructions in MARS, which also means that one cannot avoid all the delays on all the processor operating stages.

Another drawback is that during MARS encryption, some execution ports are considerably more overloaded than others. Namely, more than 78% of μoperations go either to port 0 or 1. The most overloaded is port 0, since 128 μoperations go only to this port — including 16 multiplications and extensively used rotations.

5.2 RC6

From implementers point of view, problems arising when optimizing an RC6 implementation are similar to those arising when coding MARS in many aspects: both ciphers rely on the same complex instructions, have long critical paths and overloaded

port 0. However, since RC6 uses multiplications even more extensively, it is even less parallelizable. Table 2 shows that our implementation includes 160 port 0 μoperations, which includes 40 multiplications with latency 4.

RC6 is a very Pentium II-friendly cipher, and it is very easy to code it even in the assembly language. It can also be very efficiently implemented in C: the speed difference between a C implementation and an assembly implementation is about 18%. (The difference is bigger than in the case of MARS since `gcc`, the test compiler, performs very poorly in translating the quadratic formulas of type $x \cdot (2x + 1)$ to the Pentium II assembly language.) It is straightforward to obtain an optimized assembly language implementation from the C implementation: one does not have many possibilities to reschedule the code.

5.3 Rijndael

As opposed to MARS and RC6, Rijndael [DR98] is not C-friendly (at least not `gcc`-friendly) in the sense that assembly implementation is about 44% slower than `gcc`-implementation of the same cipher. It is however mainly due to the inefficiency of the `gcc` compiler: our implementation of Rijndael makes very heavy use of the MMX technology, but also of 8-bit instructions provided by Pentium family. However, `gcc` cannot efficiently use either of these.

Rijndael can effectively use the MMX since Rijndael is based only on most simple imaginable operations (`load`, `xor`), all of which are supported by the MMX technology. Additionally, since Rijndael has large internal parallelism (at least four-times, but partially up to 16-times parallelism!), there is a large number of possibilities to reschedule its code. Our implementation was obtained by doing so in a way that all the delays in the different stages of the Pentium II operation would be minimized. The final result is very impressive for the Pentium II: it executes 2.54 μoperations per a cycle.

Not the last factor that makes Rijndael suitable for the Pentium II is the fact that almost exactly one third of the μoperations in our implementation of Rijndael go to port 2, while the remaining $2/3$ of μoperations go to ports 0 and 1. Due to this and parallelism we get that during the Rijndael encryption 3 μoperations could be executed in parallel almost all the time. However, this (not to mention other aspects like decoding and fetching delays) also makes 20 cycles per round a lower bound for Rijndael and shows that our result may be very close to the optimal one. To facilitate more efficient implementations, the Pentium II should feature three ALUs, two concurrent memory access ports and also more decoders and retirement units: features that are not cipher-specific and would improve the speed of most of the applications.

Finally, we measured the timings of r-round Rijndael for variable r without any additional fine-tuning: those implementations are unoptimized since they use the same round macros as the 10-round Rijndael without any additional effort to optimize them to reduce, say, fetching delays. In particular it turned out that 8-round Rijndael (essentially equivalent to the cipher Square [DKR97] from the implementers point of view) encrypts a block in 193 cycles. 192-bit Rijndael (12 rounds) took 286 cycles, and 256-bit Rijndael (14 rounds)—333 cycles. Note that since 12-round Rijndael is very similar to Crypton [Lim98], 286 cycles is also a (hopefully) close approximation for the speed of latter.

5.4 Twofish

Twofish is designed to be well-suited on multiple platforms, including also the Pentium II. From the implementers point of view it resembles Rijndael in many aspects, by using only simple instructions but also some large-scale components of the latter (e.g., MDS, to provide diffusion). Due to the use of low-level instructions, Twofish is also relatively slow in C compared to the assembly (the difference is about 37%).

Main difference for implementers between Rijndael and Twofish is the inclusion of the Pseudo-Hadamard Transformation that somehow complicates Rijndael's clear structure and makes it less parallelizable: while the number of μoperations in our implementation of Twofish is less than in our implementation of Rijndael, it turned out to be very difficult to use the MMX technology to optimize Twofish. Hence, Twofish is only moderately parallelizable, although the parallelism of our implementation (2.11 μoperations per cycle) is relatively good.

6 Conclusion and Work in Progress

We achieved the fastest implementations of four of the AES finalists on the Pentium II processor, obtaining speedup $8\% \ldots 26\%$ compared to the previously known implementations. Since all implementations were coded by using the same sensible assumptions, they provide a more adequate efficiency comparison of the AES finalists than the previous papers. We demonstrated that MMX can be quite efficiently used to speedup Rijndael, but is only moderately useful for other ciphers. (However, our implementations depend on the availability of MMX technology to a lesser or greater extent and in general do not run on the Pentium Pro.) We provided full specification on our time-measurement conditions to simplify for the future implementers to compare their implementations to ours.

Our implementations are not the final: we continue optimizing them. Up-to-date results will be available at the AES efficiency table [Lip99].

References

[ABK98] Ross Anderson, Eli Biham, and Lars Knudsen. Serpent: A Flexible Block Cipher With Maximum Assurance. In *The First Advanced Encryption Standard Candidate Conference*, Ventura, California, USA, 20–22 August 1998.

[BCD+98] Carolynn Burwick, Don Coppersmith, Edward D'Avignon, Rosario Gennaro, Shai Halevi, Charanjit Jutla, Stephen M. Matyas Jr., Luke O'Connor, Mohammad Peyravian, David Safford, and Nevenko Zunic. MARS — A Candidate Cipher for AES. Original paper and a tweak to it are available from `http://www.research.ibm.com/security/mars.html`, June 1998.

[DKR97] Joan Daemen, Lars Knudsen, and Vincent Rijmen. The Block Cipher Square. In Eli Biham, editor, *Fast Software Encryption '97*, volume 1267 of *Lecture Notes in Computer Science*, pages 149–165, Haifa, Israel, January 1997. Springer-Verlag.

[DR98] Joan Daemen and Vincent Rijmen. The Block Cipher Rijndael. In *Third Smart Card Research and Advanced Applications Conference Proceedings*, 1998. To appear.

[FIP77] FIPS. Data Encryption Standard. Technical report, U.S. Department of Commerce/National Bureau of Standards, National Technical Information Service, Springfield, Virginia, 1977. FIPS 46.

[Fog00] Agner Fog. How to Optimize for the Pentium Microprocessors. Available from `http://www.agner.com/assem/`, 11 March 2000.

[Gla99] Brian Gladman. AES algorithm efficiency. Unpublished. Information available from `http://www.btinternet.com/~brian.gladman/ cryptography_technology/`, January 1999.

[Int99] Intel. *Intel Architecture Optimization. Reference Manual*, 1999. Order Number 245127-001.

[Int00] Intel. *Willamette Processor Software Developer's Guide*, February 2000. Order Number 245355-001.

[Lim98] Chae Hoon Lim. Specification and Analysis of CRYPTON Version 1.0. Unpublished. Available from `http://crypt.future.co.kr/~chlim/pub/cryptonv10.ps`, 22 December 1998.

[Lip98] Helger Lipmaa. IDEA: A cipher for multimedia architectures? In Stafford Tavares and Henk Meijer, editors, *Selected Areas in Cryptography '98*, volume 1556 of *Lecture Notes in Computer Science*, pages 248–263, Kingston, Canada, 17–18 August 1998. Springer-Verlag.

[Lip99] Helger Lipmaa. AES candidates: A survey of implementations. An on-line table. Information available from `http://home.cyber.ee/helger/aes/`, January 1999.

[LM90] Xuejia Lai and James Massey. A proposal for a new block encryption standard. In I. B. Damgård, editor, *Advances in Cryptology — EUROCRYPT '90*, volume 473 of *Lecture Notes in Computer Science*, pages 389–404. Springer-Verlag, 1991, 21–24 May 1990.

[LMM94] Xuejia Lai, James L. Massey, and Sean Murphy. Markov ciphers and differential cryptanalysis. In D. W. Davies, editor, *Advances on Cryptology — EUROCRYPT '91*, volume 547 of *Lecture Notes in Computer Science*, pages 17–38, Brighton, UK, April 1994. Springer-Verlag.

[PRB98] Bart Preneel, Vincent Rijmen, and Antoon Bosselaers. Recent developments in the design of conventional algorithms. In B. Preneel, R. Govaerts, and J. Vandewalle, editors, *Computer Security and Industrial Cryptography, State of the Art and Evolution*, volume 1528 of *Lecture Notes in Computer Science*, pages 90–115. Springer-Verlag, 1998.

[Riv98] Ronald L. Rivest. Futher Notes on RC6. Unpublished. Available from `http://theory.lcs.mit.edu/~rivest/rc6-notes.txt`, 20 June 1998.

[RRSY98] Ronald L. Rivest, Matt J. B. Robshaw, R. Sidney, and Y. L. Yin. The RC6 Block Cipher. Available from `http://theory.lcs.mit.edu/~rivest/rc6.ps`, June 1998.

[SKW+99a] Bruce Schneier, John Kelsey, Doug Whiting, David Wagner, and Chris Hall. Performance comparison of the AES submissions. Unpublished. Information available from `http://www.counterpane.com/`, January 1999.

[SKW+99b] Bruce Schneier, John Kelsey, Doug Whiting, David Wagner, Chris Hall, and Niels Ferguson. *The Twofish Encryption Algorithm: A 128-Bit Block Cipher*. John Wiley & Sons, April 1999. ISBN: 0471353817.

A Pentium II for Cipher Designers and Implementers

A.1 MMX Technology

The Pentium II has 8 integer (including stack pointer) and 8 new MMX registers; the latter were not present in the Pentium Pro. While there is a great number of operations available on the integer registers, MMX registers are much more "RISCy": only a few instructions affect them, including move, Boolean operations, 16-bit arithmetic and shifts. Available set of instructions does not include several operations used in the modern block cipher design, including rotation and 32-bit multiplication. On the other hand, the MMX technology provides 64-bit versions of Boolean operations and data moves (i.e., the simplest possible operations), and also parallel 4-way addition and multiplication of 16-bit data. 16-bit multiplication is currently used in a very few ciphers, but as shown in [Lip98], ciphers that base their security on extensive use of 16-bit multiplication can be speed up considerably if using the MMX technology.

Despite of MMX's attractiveness, at the current state of the affairs many C compilers (for example, `gcc`, the standard compiler for Linux machines) do not yet produce MMX code. Hence, for the Pentium II the assembly implementations are potentially more efficient than C-language implementations. Partially by this reason, many designers and implementers of AES candidates seem not to know about MMX at all.

A.2 Processor stages.

The Pentium II processor (as other processors in the P6 family) operates in several stages. At first the instructions are fetched from the main memory and then broken down (decoded) into μoperations (simple instructions consist of only one μoperation, while complex instruction have more μoperations). Thereafter, the μoperations go via a short queue to the register allocation table that allows register renaming. After that, instructions go to reorder buffer that enables out-of-order execution. There it stays until the operands it needs are available. Ready-for-execution μoperations are sent to the execution units, and thereafter retired [Int99,Fog00]. During the optimization one has to count on all different stages of processor operation to find a good tradeoff between the delays introduced in them. The technicalities presented hereafter could be most interesting for the implementers, but also for the cipher designers who want to create ciphers optimized for the Pentium II. The most important lesson from the next is that fixing any processor stages (e.g., decoding), suitable reordering of the instructions can considerably reduce the delays at this stage. However, the same reordering usually introduces additional delays in some other stages and therefore, code reordering is always a complicated tradeoff. To achieve really fast implementations, a cipher should have great internal parallelism that provides many different instruction reordering possibilities, from what the best could be found after possibly exhaustive search. Of course, one could design a cipher that would have only one possible order of instructions, optimized specifically for Pentium II. However, such cipher could slow down severely if even slightest modifications would be introduced to the processor. Moreover, parallelism is necessary anyways, since already in the near future a processor could have dozens simultaneously working executing units.

Note that our survey is far from being complete, we refer an interested reader to [Int99,Fog00]. However, during finishing our implementations we found that also the official Pentium family optimization manual published by Intel [Int99] is far from being complete. We encountered many problems that could not have been foreseen by using only the official manuals. Often more accurate (although also not complete) information about the Pentium II was found in [Fog00]. In several places of our implementations we performed partial exhaustive search to optimally schedule the instructions. A lot of experience and luck is necessary in optimizing for Pentium II if one desires to avoid exhaustive search himself.

In-Order Decoding. Up to 3 instructions can be decoded to μoperations at time, but only the first decoder can handle instructions with more than one μoperation. It is recommended to order the instructions in the 4-1-1 sequence, which means that only every third instruction could combine in itself of more than one μoperation [Int99]. By this reason, algorithms using only "simple operations" can be potentially implemented faster than those consisting of "complex instructions". However, in some circuimstances it would also beneficial to have at least some complex instructions. Namely, if the code is properly scheduled in a way that exactly (almost) every third instruction has more than one μoperation, the decoder will feed the out-of-order execution pool with pace more than 3 μoperations per cycle. Now, if in some later stage less than 3 μoperations per cycle are fed to the execution unit (say due to the delays in fetching), this unit will not idle waiting for the next instructions from the decoder.

Instruction In-Order Fetching. The Pentium II has 16-byte internal *ifetch buffers* with the peculiarity that a new buffer is forced to start at beginning of an instruction. The first instruction of the ifetch buffer will be always decoded by decoder 0, even if the previous instruction was decoded by the same decoder and hence, other decoders would stay idle. Hence, code reordering and possible use of semantically identical instructions (in general, but not always, *shorter* instructions: for example, `mov eax,[ebx+0]` with `mov eax,[ebx]`) with different length could reduce the number of delays introduced in this stage.

Register In-Order Renaming. Pentium II has 40 hardware registers. The software registers are renamed to hardware registers after a write to (or read from) the software register. After a register has not been used for a while, it automatically retires and the next time the same register is used, a new renaming is performed. It is important to know that *only two register renamings can be done during one machine cycle*. In particular this means that generally it is beneficial to gather all instructions operating on some fixed data chunk together (i.e., to reorder the code in a suitable way). However, it is extremely difficult to detect and remove delays introduced by this stage, and therefore this stage may really become *the* bottleneck in optimization: subtle modification of code may introduce long delays in this stage. We refer to [Fog00] for more information.

Out-of-Order Execution. Pentium II has 5 execution ports (port 0, port 1, $\ldots$, port 4) that can execute instructions out-of-order. Every port has some specific meaning.

Ports 0 and 1 are ALUs (they can perform arithmetic on operands in registers), port 2 performs memory loads. Every memory write counts as two μoperations, one in port 3 (address calculation) and another one in port 4 (memory write). Up to 3 ports can execute an instruction in parallel. There are a number of arithmetic instructions that can only run in port 0 (most importantly, multiplication, rotation and integer register shifts — instructions that are widely used by some AES finalists), while some other instructions (most importantly, MMX register shifts) can only run in port 1. To obtain a throughput near to 3 μoperations per cycle, the instructions should be distributed so that no more than $2/3$ of them are arithmetic, no more than $1/3$ are memory loads and no more than $1/3$ are memory writes: a condition that is very difficult to fulfill in a practical application.

In-Order Retirement After execution, μoperations will retire in-order. During retirement, hardware registers will be written back to software registers and the μoperations leave the instruction pool. Since this is done in-order, several delays can occur, e.g., if speculative out-of-order execution of some earlier long latency instruction is not finished at the moment of retirement.