Fast Implementation of RC6 Using Intel’s SSE2 Instructions*

Xiaodong Xu
School of Computer Sciences and Technology, Beijing University of Posts and Telecommunications
Beijing 100876, China
Email: xuxiaodong2007@gmail.com   Tel.: 15811247499

ABSTRACT
RC6 is a symmetric block cipher, designed by RSA laboratory to meet the requirements of the AES competition. As one of the five AES finalists, RC6 achieves good performance with a high level of security, and especially fit for parallel processing. SSE2 is a set of Intel's instruction extensions in the IA-32's SIMD programming model. It provides the ability to perform SIMD operations on 128-bit packed integers which greatly enhances throughput and is a natural fit for RC6. In this paper we presents implementation of RC6 to fully exploit the parallel processing potential of SSE2 instructions. With specific optimization techniques, our implementation can encrypt two blocks parallelly and greatly enhances the overall performance.

Keywords: RC6, SSE2, Parallel Processing, SIMD, Instruction Level Parallelism.

1. INTRODUCTION

With more and more people doing their work through on-line applications, such as E-mail, Google Docs and E-commerce, information security becomes first rate importance. Cryptographic algorithms are designed to meet the goal that ensures confidentiality. Block ciphers, as one type of symmetric-key cipher, are used to encrypt bulk of data. Since the data transfer rates directly rely on the throughput of the algorithm, efficient implementations with high throughput are becoming increasingly important. Besides security, speed is the second main goal in designing cryptographic primitives.

RC6 is a block cipher developed to meet the requirements of Advanced Encryption Standard. As one of the five finalists of the AES candidates, RC6 has not only good security properties but also has an exceptional performance results on PC platforms. Additionally, the features of RC6 benefit significantly from Intel's instruction set extensions SSE2, which is first introduced in Pentium4, to accelerate multimedia applications such as image, speech and 3D games. These instructions provide operations on 128-bit packed integers on eight XMM registers which is useful to RC6.

In this paper we demonstrate that the speed of execution and throughput of RC6 can be increased greatly, taking full advantage of SSE2. Our implementations not only process RC6 transformations parallelly, but also encrypt/decrypt two blocks simultaneously.

The rest of the paper is organized as follows: Section 2 outlines the RC6 algorithm and SSE2 features. Section 3 presents our optimized implementation of RC6 using SSE2. Section 4 give our results benchmarked by clock cycles per block and throughput bits per second. Section 5 concludes the paper and in the final section, gives a future outlook.

2. AN OUTLINE FOR RC6 AND SSE2

2.1. The RC6 Block Cipher

The RC6 block cipher consists of three components: key expansion algorithm, encryption algorithm and decryption algorithm. We only describe RC6 Encryption algorithm here, for a detailed descriptions of RC6, please refer to [1].

Notations and primitive operations in RC6:

\[ \lg(x) \]: base-two logarithm of \( x \)
\[ + \]: Two's complement addition of words
\[ \text{xor} \]: Bit-wise exclusive-OR of words
\[ <<< \]: Left-rotation of words, the cyclic rotation of words \( x \) left by \( y \) is denoted \( x<<<y \).

All the above operations are modulo word size.

The RC6 Encryption Algorithm:

**Input**: Plaintext stored in four \( w \)-bit input registers \( A,B,C,D \);
Number of \( r \) of rounds; \( w \)-bit round keys \( S[0,...,2r+3] \)

**Output**: Ciphertext stored in \( A,B,C,D \)

**Procedure**:

\[ B = B + S[0] \] step 1
\[ D = D + S[1] \] step 2

for \( i = 1 \) to \( r \) do

\[ t = (B*(2B+1))<<<\lgw\] step 3
\[ u = (D*(2D+1))<<<\lgw\] step 4
\[ A = ((A \text{xor} t)<<<i) + S[2i]\] step 5
\[ C = ((C \text{xor} u)<<<i) + S[2i+1]\] step 6
\[ (A,B,C,D) = (B,C,D,A) \] step 7

\[ A = A + S[2r+2] \] step 8
\[ C = C + S[2r+3] \] step 9

In our implementation, the word size \( w = 32 \), and the iteration round \( r = 20 \).

2.2. Intel's SSE2 Extensions Overview
SSE2 was first introduced in Pentium4 and Xeon processors to enhance the performance of multimedia applications, such as 3-D graphics and video decoding. SSE2 extends the SIMD model with support for packed double-precision floating-point values and for 128-bit packed integers. We only care about integer instructions since operations of RC6 are all integer arithmetic.

Take two instructions for example:

- **paddd**: Adds the 4 signed or unsigned 32-bit integers in XMM2 to the 4 signed or unsigned 32-bit integers in XMM1, **Figure1**.

- **pshufd**: Shuffles the 4 signed or unsigned 32-bit integers in XMM2 to the 4 signed or unsigned 32-bit integers in XMM1, as specified by imm8, **Figure2**.

**Figure1**

**Figure2**

3. OUR IMPLEMENTATION

3.1. Block Level Parallelism

From section 2.1, we observe that the incoming 128-bit plaintext block (seen as four sequential words A,B,C,D) is not fully processed in parallel. Instead, A and C are processed parallelly, so are B and D. So packing the four plaintext words into one 128-bit XMM register can not benefit from SSE2 instructions. But packing two words into one XMM register do not fully exploit the advantage of SSE2 since SSE2 enables four word integers processed parallelly. So why not process two blocks at a time! Given two blocks A1:B1:C1:D1 and A2:B2:C2:D2, through a series of pack and unpack operations, we get A1:C1:A2:C2 and B1:D1:B2:D2, which can be store in two XMM registers and processed efficiently. After the whole encryption process, we pack them back to A1:B1:C1:D1 and A2:B2:C2:D2 and output the two encrypted blocks. This trick makes our implementation speed almost double.

The following shows the packing process.

**Input**: A1:B1:C1:D1, A2:B2:C2:D2


**Implement**:  

- movdqa xmm0, plaintBlock1 ; xmm0: A1:B1:C1:D1  
- movdqa xmm1, plaintBlock2 ; xmm1: A2:B2:C2:D2  
- movdqa xmm2, xmm0  
- punpckldq xmm0, xmm1 ; xmm0: A1:A2:B1:B2  
- punpckhdq xmm2, xmm1 ; xmm2: C1:C2:D1:D2  
- movdqa xmm1, xmm0  
- punpckldq xmm0, xmm2 ; xmm0: A1:B1:C1:D1  
- punpckhdq xmm1, xmm2 ; xmm1: B1:B2:D1:D2

After the whole encryption is finished, we should pack A1:A2:C1:C2 and B1:B2:D1:D2 back to A1:B1:C1:D1 and A2:B2:C2:D2, thus we get two encrypted blocks.

3.2. Operation Level Parallelism

After two plaintext blocks are packed and reshuffled, the following task is to translate basic arithmetic to parallel SIMD code. As RC6 possesses a good parallel structure, we can properly arrange the steps so that some of them can be processed parallelly. According to section 2.1, it is easy to notice that step 1 and step 2, step 3 and step 4, step 5 and step 6, step 7 and step 9, can be executed parallelly. Our implementation is described in the following in detail. For simplicity and straightforwardness, Intel’ Intrinsics for SSE2 are used instead of raw assembly code. For a detailed description of Intrinsics, please reference to [6].

Our Implementation is as follows:

**Notations and definitions**:
- iOnes: 128-bit constant integer, (1,1,1,1)
- imusk: 128-bit constant integer, (word(-1),0,word(-1),0)

**step 1 and step 2**:

\[
\text{pshufd}(S[0], S[1], 0x44) \rightarrow S_0,S_1,S_2,S_3; \text{ pack the first two words of round key array into 128-bit XMM register};
\]

\[
\text{padd}(B_1,D_1,B_2,D_2, S_0,S_1,S_2,S_3); \text{ add packed four words parallelly}.
\]

**step 3 and step 4**:

\[
\text{pslld}(B_1,D_1,B_2,D_2, 1) \rightarrow \text{temp128: logically shift left by one bit, so each four words multiply};
\]

\[
\text{padd}(\text{temp128}, iOnes) \rightarrow \text{temp128: add 1 for each four words parallelly};
\]

Since there’s no packed multiply, we have to multiply respectively.

\[
\text{pand}(\text{pmuludq}(B_1,D_1,B_2,D_2), \text{temp128}), \text{imusk}) \rightarrow \text{low128: calculate } D_1*(D_i+1) \text{ and } D_2*(D_2+1), \text{ and we mask the higher part of the two 64-bit integer results};
\]

\[
\text{pand}(\text{pmuludq}(\text{psrlq}(B_1,D_1,B_2,D_2,4),\text{psrlq}(\text{temp128},4)),\text{imusk}) \rightarrow \text{high128: calculate } B_1*(B_1+1) \text{ and } B_2*(B_2+1), \text{ and we mask the higher part of the two 64-bit integer results};
\]

\[
\text{por}(\text{low, pslldq}(\text{high},4)) \rightarrow \text{temp128: shift high by 4 bytes and}
\]
xor with low bit by bit, then we get the value of \( f(x) = x^*(2x+1) \) for \( B_3, D_1, B_2, D_2 \) respectively.

\[
p(xor(temp128, 5), \ psll(temp128, 32-5)) \rightarrow temp128: \text{rotate by lg32.}
\]

step 5 and step 6:

\[
pxor(A_1 : C_1 : A_2 : C_2), \ temp128) \rightarrow \ A_1 : C_1 : A_2 : C_2; \text{bit by bit xor}
\]

movdqa(word temp1[4], temp128), movdqa(word temp2[4], A_1 : C_1 : A_2 : C_2); store the two 128-bit values since there's no packed rotate, we have to do it manually.

\[
\text{rotl(temp2[3]), temp1[2]);}
\]

movdqa(A_1 : C_1 : A_2 : C_2, temp2[2]): save back to A_1 : C_1 : A_2 : C_2

pshufd(S[2i] S[2i+1], 0x44) —— S: pack current round key S[2i] and S[2i+1] into XMM register;

paddd(A_1 : C_1 : A_2 : C_2, S[2i] S[2i+1]): add packed four 32-bit words parallelly.

step 7:

Step 7 is a permutation \((A, B, C, D) = (B, C, D, A)\), for our two blocks, that is:

\[
A_1 : C_1 : A_2 : C_2 = B_1 : D_1 : B_2 : D_2 = C_1 : A_1 : C_2 : A_2
\]

movdqa(temp3, B_1 : D_1 : B_2 : D_2) = pshufd(A_1 : C_1 : A_2 : C_2, 0xb1)

A_1 : C_1 : A_2 : C_2 = temp3

step 8 and step 9:

pshufd(S[2i+2], S[2i+3], 0x44) —— S_0 : S_1 : S_2 : S_3 : S_4: pack the last two words of round key array S[2i+2] and S[2i+3] into XMM registers;

paddd(B_1 : D_1 : B_2 : D_2, S_0 : S_1 : S_2 : S_3 : S_4): add packed four 32-bit integer.

3.3. Instruction Level Parallelism

Though we process the encryption arithmetic as parallelly as possible, there's still space to improve. The Pentium processor allows to execute two instructions in parallel through two five-stage pipelines, called U pipeline and V pipeline. This instruction pairing is carried out automatically and independently for programmers. However, that does not mean CPU will do it all. Instead, to pair two instructions, there's several constraints are to be satisfied. So it's up to programmers to organize the instructions elegantly, so that a better performance due to the superscalar structure is achieved. The following is pairing rules for Intel's SIMD instructions:

- The second instruction does not read or write a register which the first instruction writes to.

- MMX/SSE shift, pack or unpack instructions can execute in either pipe but cannot pair with other MMX/SSE shift, pack or unpack instructions.

- MMX/SSE multiply instructions can execute in either pipe but cannot pair with other MMX/SSE multiply instructions. They take 3 clock cycles and the last 2 clock cycles can overlap with subsequent instructions.

- An MMX/SSE instruction which accesses memory or integer registers can execute only in the U-pipe and cannot pair with a non-MMX/SSE instruction.

<table>
<thead>
<tr>
<th>Instructions(S)</th>
<th>C</th>
<th>Instructions(R)</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>movdqa xmm6, xmm1</td>
<td>1</td>
<td>movdqa xmm6, xmm1</td>
<td>1</td>
</tr>
<tr>
<td>psll xmm6, 1</td>
<td>2</td>
<td>7 movdqa xmm2, xmm1</td>
<td>P</td>
</tr>
<tr>
<td>paddd xmm6, iOnes</td>
<td>1</td>
<td>2 paddd xmm6, xmm1</td>
<td>1</td>
</tr>
<tr>
<td>movdqa xmm7, xmm6</td>
<td>1</td>
<td>8 pslldq xmm2, 4</td>
<td>2P</td>
</tr>
<tr>
<td>psrldq xmm6, xmm1</td>
<td>3</td>
<td>3 paddd xmm6, iOnes</td>
<td>1</td>
</tr>
<tr>
<td>padd xmm6, imusk</td>
<td>1</td>
<td>4 movdqa xmm7, xmm6</td>
<td>1</td>
</tr>
<tr>
<td>movdqa xmm2, xmm1</td>
<td>1</td>
<td>5 pslldq xmm6, xmm1</td>
<td>3</td>
</tr>
<tr>
<td>paddd xmm7, 4</td>
<td>2</td>
<td>9 pslldq xmm7, 4</td>
<td>2</td>
</tr>
<tr>
<td>pslldq xmm7, 4</td>
<td>2</td>
<td>6 padd xmm6, imusk</td>
<td>1</td>
</tr>
</tbody>
</table>

Total: 14 Cycles  
Total: 9 Cycles

C: cycle, P: paired, O: overlapped with subsequent instruction  
S: straightforward implemenation, R: re-arranged implementation

We take a code segment in assembly to illustrate how to optimize the code. In Table 1, the left two columns are straightforward implementation, while the right two columns are optimized implementation after rearranging. The instructions are labeled by the number before them.

Let's take a closer scrutiny to the code.

Instruction 2 uses psll to multiply four integer words. It can be replaced by paddd to achieve the same results. Since the latter consumes only 1 cycle but the former consumes 2, we save 1 cycle by doing so.

Multiply instructions are quite time-consuming. According to rule 3, after issuing a multiply instruction, wait until three clocks later before using the result. So the overall throughput can be optimized to 1. In this guideline, we place instruction 9 after 5. Since 9 does not use the register which 5 writes to, its two execution cycles can overlap exactly with the last two cycles of 5. This saves two cycles.

Besides, it's easy to notice that instructions 1-4 have high data dependency. We move 7 to line 2, 8 to line 4 to remove data dependency, so that 1 and 7, 2 and 8 can be paired and this saves two cycles in overall.

After the above optimization, we save 5 cycles out of 14 and achieve up to 56% enhancements in performance.

4. OUR RESULTS

We test the cycles and throughput on Intel Core2 Duo™ 3.0G PC, as Table 2 illustrated.
5. CONCLUSIONS

The goal of our work is to show how to speed up the RC6 block cipher utilizing Intel's SSE2 128-bit SIMD Integer Instructions. The case study of RC6 provides a way to other block ciphers. By way of three levels of parallelism (block level parallelism, operation level parallelism and instruction level parallelism), this goal has been successfully met.

6. FUTURE WORK

The optimization of RC6 that we have followed can be taken even further.

(1) The data movement between CPU and memory can be speeded up by using data prefetching. In this way the data required next can be brought into the cache in advance.

(2) The data can be aligned to 16 bytes prior to loading it into the processor. This increases the memory performance enhancement leading to a further improvement in speed.

(3) GPU offers a tremendous amount of computational bandwidth that was until now largely unusable for cryptographic computations due to a lack of integer arithmetic and user-friendly APIs. The latest generation of GPUs, which introduced integer/binary arithmetic, has been leveraged to create several implement. Two mainstream GPU camps, NVIDIA and ATI, both introduced their general-purpose GPU computing model, CUDA and ATI Stream respectively. They are designed to handle processing massive amounts of data efficiently and hold the potential for significant speedup for data parallel problems, basically for 5-10 times faster. Exploiting GPU parallel computing technology to cryptographic applications may be rewarding.

7. REFERENCES


Xu Xiaodong is a Postgraduate of Information Security Center in School of Computer Science and Technology of Beijing University Posts and Technology. He obtained his bachelor's degree in mathematics from Wuhan University of Science and Technology in 2006, and work as a software programmer in CS&S. His research interests are mainly on implementation of cryptographic algorithms.