情報保護學會論文誌
第18卷 第6(A)號, 2008. 12

64-비트 프로세서에서 AES 고속 구현

정 창 호†‡, 박 일 환
한국전자통신연구원 부설연구소

High Speed AES Implementation on 64 bits Processors

Changho Jung†‡, Ilhwan Park
The Attached Institute of ETRI

요 약

본 논문은 최근 많이 사용되는 64-비트 프로세서인 Intel Core2 프로세서와 AMD Athlon64 프로세서에서 AES 알고리즘을 고속 구현하는 기법을 제시한다. 먼저 EM64T 아키텍처의 Core2 프로세서는 메모리 접근 명령어 처리 효율이 연산 명령어 처리 효율보다 떨어진다. 때문에 메모리 접근 명령어의 비율이 높게 구성된 기존 AES 구현기법은 메모리 병목현상이 발생된다. 이에 메모리 접근 명령어 비율을 낮춘 부분 라운드키 기법을 제시한다. ECB 모드로 구현한 결과 Core2Duo 3.0 Ghz 프로세서에서 185 cycles/block, 2.0 Gbps의 성능을 보여주었다. 이 결과는 가장 빠르다고 알려진 bernstein 코드보다 35 cycles/block 빠르다. 한편 AMD64 아키텍처의 Athlon64 프로세서에서는 명령어 디코딩 과정에서 발생하는 병목현상을 제거하므로써 속도를 향상시켰다. 그 결과 Athlon64 프로세서에서 170 cycles/block의 성능을 나타났다. 이는 가장 빠르다고 알려진 Matsui의 비공개 코드와 성능이 동일하다.

ABSTRACT

This paper suggests a new way to implement high speed AES on Intel Core2 processors and AMD Athlon64 processors, which are used all over the world today. First, Core2 Processors of EM64T architecture's memory-access-instruction processing efficiency are lower than calculus-instruction processing efficiency. So, previous AES implementation techniques, which had a high rate of memory-access-instruction, could cause memory-bottleneck. To improve this problem we present the partial round key techniques that reduce the rate of memory-access-instruction. The result in Intel Core2Duo 3.0 Ghz Processors show 185 cycles/block and 2.0 Gbps's throughputs in ECB mode. This is 35 cycles/block faster than bernstein software, which is known for being the fastest way. On the other side, in AMD64 processors of AMD64 architecture, by removing bottlenecks that occur in decoding processing we could improve the speed, with the result that the Athlon64 processor reached 170 cycles/block. The result that we present is the same performance of Matsui's unpublished software.

Keywords : AES, 64-bits, Core2, AMD64, Assembly Code

Ⅰ. 서 론

최근 들어 32-비트와 64-비트 명령어를 호환하는 프로세서의 등장으로 64-비트 프로세서 시대를 열어가고

접수일 : 2008년 7월 7일; 수정일 : 2008년 9월 29일
채택일 : 2008년 10월 24일
† ‡ 주저자, 교신저자 : zangho@ensec.re.kr

있다. 얼마 전만 해도 대다수의 컴퓨터가 32-비트 프로세서를 대표하는 x86 프로세서를 탑재하고 있었기 때문에 x86 프로세서에서의 고속 구현은 많은 관심을 불러 모았다. 특히 암호 알고리즘의 속도를 비교할 때 x86 프로세서에서의 측정 속도가 중요한 척도가 되었다. 그러나 근래에 64-비트 프로세서에 대한 관심의 증폭으로 32비트 최적화 연구는 64-비트 최적화 연구로 서서히 옮겨가고 있다.

최근 많이 사용되는 64-비트 프로세서인 Intel Core2 프로세서와 AMD Athlon64 프로세서는 범용 레지스터가 16 개로 x86 프로세서보다 두 배 증가했다는 점에서 큰 장점을 가지고 있다. 가령 x86 프로세서에서는 8 개의 범용 레지스터로 AES를 구현하려면 레지스터가 부족해서 중간 데이터를 반드시 스택 메모리나 MMX 레지스터에 저장하고 불러오는 부가적인 처리시간이 소요되었던 반면 64-비트 프로세서에서는 이러한 오버헤드를 제거할 수 있게 되었다.

본 논문은 AES 알고리즘을 64-비트 프로세서에서 기존의 구현 방법보다 빠르게 동작하도록 구현하는 방법을 제시한다. 64-비트 프로세서에는 Intel의 EM64T 아키텍처를 기반으로 하는 Core2 프로세서와 AMD의 AMD64 아키텍처를 기반으로 하는 Athlon64 프로세서가 있는데, 아키텍처 기반이 다른 두 프로세서는 동일한 64-비트 명령어를 처리할 수 있지만 내부 동작방식에서 차이가 있기 때문에 고속 구현 방법과 결과에도 차이가 있다. Core2 프로세서에서 부분 라운드키 기법을 적용하여 고속 구현한 결과 185 cycles/block의 성능을 보여주었다. 한편 AMD64 프로세서에서는 명령어 코드 크기를 최소화하는 기법을 적용하여 170 cycles/block의 속도에 도달하였다. 이 결과들은 현재 64-비트 프로세서에서 구현된 AES 중 가장 빠르다.

본 논문은 다음과 같이 구성된다. 2장에서는 64-비트 프로세서에서 AES 고속 구현에 관한 발표된 기존 연구들을 살펴보고, 3장에서는 AES 알고리즘과 64-비트 프로세서인 EM64T 프로세서와 AMD64 프로세서의 차이를 살펴본다. 4장과 5장에서는 각각 EM64T 프로세서와 AMD64 프로세서에서 고속 구현한 연구 결과를 상세히 설명한다. 마지막 6장에서는 64-비트 프로세서에서 AES 고속 구현의 이론적 한계에 대해 논의하고 결론을 맺는다.

Ⅱ. 관련 연구

'AES Speed'[1]는 다양한 AES 소프트웨어 구현의 속도결과를 정리해놓고 있다. 이 문서를 참고로 x86 프로세서에서 구현한 AES 구현 속도를 비교하면 [표 1]과 같다. Intel Pentium 4에서 AES를 가장 빠르게 구현한 것은 Dag Arne Osvik[2]이 구현한 것으로 240 cycles/ block으로 구현하였다. 반면, 암호 라이브러리로 많이 알려진 OpenSSL[5]의 경우 AES를 가장 빠르게 구현한 버전은 현재 0.9.8b 버전으로 구현 속도는 352 cycles/block이며 가장 빠른 구현과 비교하면 50% 가까이 느리다.

[표 1] x86 프로세서에서 구현한 AES 구현 속도 비교

소프트웨어	속도 (cycles/block)	CPU
Dag Arne Osvik[10]	224	Intel Pentium 3
Dag Arne Osvik[2]	240	Intel Pentium 4
Helger Lipmaa[3]	254	Intel Pentium 4
Mitsuru Matsui[4]	256	Intel Pentium 4
Bernstein[1,8]	271	Intel Pentium 4
OpenSSL 0.9.8b[5]	352	Intel Pentium 4

반면 64-비트 Intel Core2 프로세서에서 가장 빠른 AES 구현은 Matsui의 bitslice구현 코드[6]이다. 그는 2048-바이트 AES Bitslice를 147 cycles/block으로 구현하여 연구 결과를 인정받았으나 실제 적용시에는 bitslice 입력구조로 변환하기 위한 부가적인 연산이 필요하며, 2048-바이트 단위로만 암호화할 수 있기 때문에 용도의 제한이 많고 범용 AES primitive로 사용하기엔 성능저하가 유발된다. 다음 순위로는 Hongjun wu의 구현 기법[7,8]으로 CTR 모드 전용으로 최적화하여 구현하였다. CTR 모드의 특성상 Counter의 마지막 바이트를 제외한 15 바이트 평문은 256 번 동안 변하지 않는다. 따라서 평문의 15 바이트가 동일한 경우의 1 라운드의 출력과 2 라운드의 출력을 미리 계산해서 반복 사용할 수 있게 된다. Hongjun wu는 256 번 마다 고정된 15 바이트에 대한 1 라운드 출력과 2 라운드 출력을 미리 계산해 놓고, 1 라운드 연산시 1 개 명령어, 2 라운드 연산시 4 개 명령어로만 구현하여 속도를 향상시켰다. 이 기법은 모든 알고리즘의 CTR모드에 적용할 수 있는 기법으로 유용하지만, 단지 CTR 모드에서만

속도향상이 있을 뿐 다른 운영모드의 AES 구현에서 적용할 수 있는 고속화 요소가 없다. 다음 순위에 있는 Bernstein(amd64-2)[1,8]이 구현한 AES는 어셈블리어로 구현하여 220 cycles/block으로 동작하며 모든 운영모드에서 활용 가능한 가장 빠른 AES 소프트웨어이다. 본 논문은 Matsui의 bitslice구현 기법과 Hongjun wu의 구현 기법은 범용 AES primitive로 볼 수 없기 때문에 성능비교에서 제외하였다.

[표 2] Intel Core2Duo에서 64비트 코드로 구현한 AES 구현 속도 비교

소프트웨어	속 도 (cycles/block)	운영모드
Mitsuru Matsui[6]	147	128 비트 bitslice
Hongjun wu[7,8]	202	CTR 모드 전용
Bernstein(amd64-2) [1,8]	220	CTR 모드
Mitsuru Matsui[4]	232	ECB 모드
Brian Gladman[8,9]	288	CTR 모드
OpenSSL 0.9.8e[5]	302	ECB 모드

다음 [표 3]은 AMD Athlon64에서 64-비트 코드로 구현한 AES 구현 소프트웨어들의 속도를 비교하였다. AMD Athlon64 프로세서에서 가장 빠른 AES 구현은 Matsui의 64-비트 어셈블리어 구현 코드[4]로 범용 AES이며, Intel Core2 프로세서보다 빠르게 구현하는데 성공하였다. 이는 Athlon64 프로세서의 파이프라인 효율성을 극대화한 결과로 평가되나, 코드가 공개되지 않아 실제 구현기술의 노하우는 밝혀지지 않은 상태이다.

[표 3] AMD Athlon64에서 64-비트 코드로 구현한 AES 구현 속도 비교

소프트웨어	속도 (cycles/block)	운영모드
Mitsuru Matsui[4]	170	ECB 모드
Helger Lipmaa[3]	199	ECB 모드
Bernstein(amd64-1) [1,8]	214	CTR 모드

범용 AES로 구현된 소프트웨어들의 구현 기술을 비교해보면 [표 4]와 같다. 우선 암호화에 사용된 치환 테이블의 크기는 2, 4, 4+4 KB로 차이가 나타난다. 치환 테이블은 일반적으로 Gladman의 구현과 같이 1 KB의 치환 테이블 네 개로 구성된 4 KB로 구현되지만, Matsui는 마지막 라운드 고속화를 위해 마지막 라운드용 치환 테이블 4 KB를 추가로 사용하였다. Bernstein과 OpenSSL은 4 KB의 치환 테이블에서 반복되는 부분을 제거하여 2 KB만 가지고 구현하였다. 실제 실험을 통해 비교한 결과 2 KB 치환 테이블로 구현한 AES와 4 KB 치환 테이블로 구현한 AES의 속도에는 차이가 없었으나 4+4 KB 치환 테이블로 구현한 AES는 마지막 라운드에서 약간의 속도 향상의 이점이 있었다.

한편 메모리 접근 회수는 라운드마다 치환 테이블 참조 16회, 라운드키 읽기 4회가 반드시 사용되어서 라운드당 최소 20회가 반드시 사용되었으며, 라운드당 명령어의 수는 40개 이하로 줄이기에는 한계에 도달한 것으로 판단된다. 대체로 AES는 메모리 접근 명령어의 비중이 전체 명령어수의 50%에 해당되며, 메모리 접근 명령어 처리가 느린 Intel EM64T 프로세서에서는 고속구현에 불리한 점으로 작용한다.

[표 4] 64비트 코드로 구현한 범용 AES 구현 소프트웨어의 구현 기법 비교

소프트웨어	치환 테이블 크기 (KB)	메모리 접근 명령어 (회/라운드)	명령어수 (개/라운드)
Bernstein(amd64-1)	2	20	44
Bernstein(amd64-2)	2	20	40
Mitsuru Matsui	4+4	20	44
Brian Gladman	4	20	60
OpenSSL 0.9.8e	2	26	54

Ⅲ. 64-비트 프로세서에서 AES 고속구현 기술

여기서는 AES 알고리즘의 고속구현 기법의 이해를 돕기 위한 AES 알고리즘의 64-비트 코드구조를 간단히 설명하고 64-비트 프로세서의 두 종류인 EM64T 아키텍처와 AMD64 아키텍처의 차이점을 비교하며 고속화에 있어서 고려해야할 사항을 살펴보자.

3.1 AES 알고리즘

AES의 반복되는 라운드에는 Subbytes, Shiftrows, Mixcolumns와 Addroundkey로 구성되어 있다. 한 라운드는 8비트 입력-32비트 출력 치환 테이블 T_0, T_1,

T_2, T_3를 이용하면 [표 5]와 같이 간단히 구현할 수 있다. 각 라운드의 입력값은 4개의 워드 A, B, C, D로 구성되어 있으며, 각 워드는 A_0, A_1, A_2, A_3와 같이 4 바이트로 구성되어 있다. 각 라운드의 라운드키는 4 바이트 워드 rkA, rkB, rkC, rkD로 구성되어 있다. 16개의 모든 입력 바이트는 각각의 치환 테이블 참조 값을 출력하고 그 출력 결과는 라운드키와 함께 xor를 수행한다.

[표 5] AES 한 라운드의 코드

A' = T0[A_0] ^ T1[B_1] ^ T2[C_2] ^ T3[D_3] ^ rkA
B' = T0[B_0] ^ T1[C_1] ^ T2[D_2] ^ T3[A_3] ^ rkB
C' = T0[C_0] ^ T1[D_1] ^ T2[A_2] ^ T3[B_3] ^ rkC
D' = T0[D_0] ^ T1[A_1] ^ T2[B_2] ^ T3[C_3] ^ rkD

[표 5]에서 표현한 한 라운드와 동일하게 64-비트 어셈블리 코드로 표현하면 [표 6]과 같은 코드를 4번 반복하여 구성할 수 있다. [표 6]은 입력 워드 A에 대한 Subbytes, Shiftrows, Mixcolumns를 처리하는 템플릿 코드이며, 나머지 B, C, D 워드에 대해서도 동일한 구조로 구현할 수 있다. Addroundkey는 메모리에 저장된 라운드키와 4번의 xor연산으로 구현한다.

[표 6] AES 한 라운드의 한 워드를 처리하는 64-비트 어셈블리 코드

```
movzx     rsi, al                ; A0
mov/xor   register0, T0[rsi*4]   ; A'^=T0[A0]
movzx     rsi, ah                ; A1
mov/xor   register3, T1[rsi*4]   ; D'^=T1[A1]
shr       eax, 16
movzx     rsi, al                ; A2
mov/xor   register2, T2[rsi*4]   ; C'^=T2[A2]
movzx     rsi, ah                ; A3
mov/xor   register1, T3[rsi*4]   ; B'^=T3[A3]
```

3.2 64-비트 프로세서

64-비트 프로세서의 두 종류인 Intel의 EM64T와 AMD의 AMD64의 아키텍처에는 차이점이 있다. 먼저 AMD64 프로세서는 3-way superscalar 프로세서로 한 사이클마다 세 개의 명령어를 처리할 수 있으며, 한 사이클당 두 개의 메모리 접근 명령어를 포함할 수 있다. 가령 메모리 접근 명령어가 연속에서 3번 배열하면 한 사이클에 세 개의 명령어를 처리할 수 없게 된다. 또 다른 특징으로 파이프라인의 패치 단계에서 한 사이클당 16 바이트의 명령어 코드를 디코딩할 수 있다. 따라서 연속된 명령어 3개의 코드 크기가 16바이트를 초과하면 디코딩과정에서 병목현상이 발생하여 한 사이클에 세 개의 명령어를 처리할 수 없다.

EM64T 프로세서의 경우 AMD64와 달리 디코딩된 명령어는 cache에 저장되어 디코딩 용량을 확장하였다. 따라서 명령어의 코드 크기에 의한 병목현상이 나타나지 않는다. 하지만 사이클당 하나의 데이터 읽기와 쓰기 명령어만 처리할 수 있어서, 데이터 접근 명령어가 연속해서 배열되면 명령어 처리량이 감소하게 된다.

[표 7]은 EM64T 프로세서와 AMD64 프로세서 테스트용 예제 코드이다. EM64T 프로세서의 latency를 비교해보면 메모리 접근 명령어의 개수에 비례하는 것을 발견할 수 있다. 반면 AMD64 프로세서는 명령어의 길이가 16 바이트를 넘지 않으면 3-way superscalar 특성으로 3 Instruction/cycle로 동작함을 알 수 있다.

[표 7] EM64T 프로세서와 AMD64 프로세서 테스트용 예제 코드

코 드	명령어 코드 크기 (bytes)	EM64T의 latency (cycle)	AMD64의 latency (cycle)
<코드 1> xor rax, 0[rsi+rcx] xor rbx, 8[rsi+rcx] add rcx, 16	13	2.2	1
<코드 2> xor rax, table+0[rcx] xor rbx, table+8[rcx] add rcx, 16	18	2.2	1.4-1.9
<코드 3> movzx rcx, al xor rbx, table+[rcx*8] shr eax, 8	15	1.7	1

Ⅳ. EM64T 프로세서에서 고속 구현

앞에서 살펴본 바와 같이, EM64T 프로세서에서는 메모리 접근 횟수만큼은 사이클이 소모되게 된다. 특히 메모리 접근 횟수의 비중이 50%에 가까운 AES를 구현할 땐, 메모리 접근 명령어를 잘 분산시킨다면 메모리

접근 횟수만큼의 사이클로 구현할 수 있다. 수식 (1)과 같이 AES 알고리즘은 한 블록을 암호화하는 과정에서 메모리 접근 횟수는 총 212회이다.

$$\begin{array}{l} 4+4\text{회(평문읽기,암호문쓰기)} \\ +\ 10\times 16\text{회(치환테이블참조)} \\ +\ \ 11\times 4\text{회(라운드키읽기)} \\ \hline 212\text{회} \end{array} \quad (1)$$

Matsui는 이를 근거로 Pentium 4에서 AES를 구현하려면 적어도 212 cycles/block이 소모된다고 주장했다[4]. 한편으로는 Bernstein(amd64-2)이 구현한 코드는 CTR 모드 구현을 포함하여 220 cycles/block에 동작하는데, Matsui의 주장이 이론적인 한계 속도라고 한다면 Bernstein의 구현에 의해 거의 한계점에 도달한 것으로 보인다. 그러나 메모리 접근 횟수를 줄일 수 있다면 이론적인 한계 속도는 수정될 수 있으며, Bernstein의 구현보다 빠르게 동작하는 AES 구현은 가능할 것이다. 다음 절에서 부분 라운드키를 이용한 고속구현 기법을 소개하고, 실제 구현에 따른 기술들과 실험 결과를 살펴보자.

4.1 부분 라운드키를 이용한 고속구현 기법

이 연구는 메모리 접근 횟수를 최소화하여 AES의 속도향상을 시도하고자 했다. 메모리 접근은 수식 (1)과 같이 평문읽기, 치환테이블참조, 라운드키읽기에서 발생하는데 제안하는 고속구현 기법은 라운드키 읽기 횟수를 44회에서 14회로 감소하여 AES의 속도 향상을 가능하게 하였다.

제안하는 고속구현 기법을 설명하기에 앞서 AES의 라운드키 생성과정을 [표 8]을 통해 간단히 살펴보자. [표 8]에서 4 바이트 라운드키 $rkA_i(1 \le i \le 10)$는 i 번째 라운드의 첫 번째 워드 라운드키를 의미한다. rkA_i는 Rotate 연산, Subbytes 연산 그리고 두 번의 xor 연산을 통해 생성한다. 반면 rkB_i, rkC_i, rkD_i는 각각 한 번의 xor연산으로 생성한다.

[표 8] AES 라운드키 생성과정

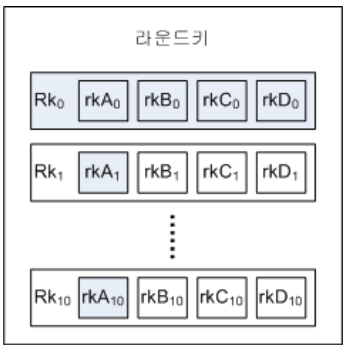
rkA_i = Subbytes(Rotate(rkD_{i-1})) xor rconti xor rkA_{i-1}
rkB_i = rkA_i xor rkB_{i-1}
rkC_i = rkB_i xor rkC_{i-1}
rkD_i = rkC_i xor rkD_{i-1}

일반적으로 AES를 구현할 때, 라운드키 생성부분과 암호화 과정부분을 분리하여 구현한다. 라운드키 생성부분에서는 라운드키를 생성하여 메모리에 저장하고, 암호화 과정부분은 미리 생성된 라운드키를 메모리로부터 읽어온다. 한 블록 암호화 과정중 라운드키를 읽어오기 위한 메모리 접근은 총 44회 발생된다.

메모리 접근 횟수를 최소화하는 관점에서 128 비트 마스터 키인 rkA_0, rkB_0, rkC_0, rkD_0만 메모리에서 읽어오면 나머지 라운드키를 생성하여 암호화 과정을 구현할 수 있다. 하지만 라운드키를 생성하는 과정중 rkA_1, rkA_2, … , rkA_{10}을 생성시 계산량이 많고 Subbytes마다 4번의 치환 테이블 참조가 발생하여 결국 메모리 접근 횟수가 44회가 되므로 메모리 접근 횟수는 줄어들지 않는다.

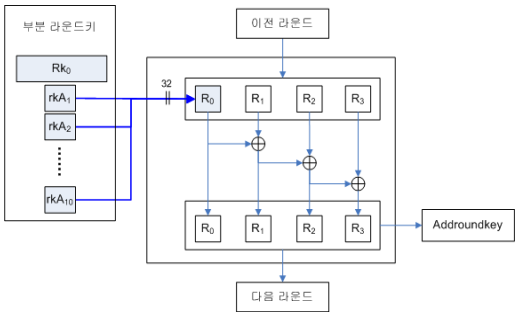
메모리 접근 횟수를 최소화하는 관점에서 또 다른 방법은 마스터 키 4워드와 계산량이 많은 rkA_1, rkA_2, …, rkA_{10} 10워드를 메모리에서 읽어오고 나머지 라운드키를 생성하여 암호화 과정을 구현하는 방법이다. rkA_i만 메모리로부터 읽어온다면 나머지 라운드키 rkB_i, rkC_i, rkD_i는 세 번의 xor로 생성할 수 있다. 따라서 메모리 접근 횟수는 14회이고, 나머지 라운드키를 생성하기 위해 30번의 xor연산만 필요하다. 다만 추가적으로 4개의 레지스터를 확보할 수 있어야 하는데 64-비트 프로세서에서 충분한 레지스터를 이용하여 이와 같이 구현한다면, 메모리 접근 횟수는 14회로 최소화시킬 수 있다.

본 논문이 제안하는 AES 고속구현 방법은 라운드키의 일부만 메모리로부터 읽어오는데, 이 때 메모리에서 읽어오는 라운드키를 부분 라운드키라고 한다. 부분 라운드키는 [그림 1]과 같이 44 워드 중 회색 박스부분인 14 워드 Rk_0와 rkA_1~rkA_{10}로 구성되어있다.

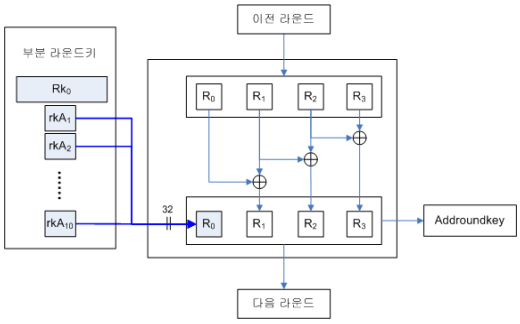


[그림 1] 전체 라운드키와 부분 라운드키

[그림 2]를 보면, 부분 라운드키의 첫 번째 4 워드 라운드키 Rk_0는 암호화 시작시 레지스터 R_0~ R_3에 읽어와서 초기화한다. 나머지 10 워드 라운드키 rkA_i는 매 라운드마다 1번씩만 레지스터 R_0에 업데이트하여 나머지 라운드키 rkB_i, rkC_i, rkD_i를 생성하게 된다. 기존 구현 방법이 한 라운드마다 라운드키 읽기를 4회 실행하였으나, 제안하는 구현 방법은 라운드키 읽기 1회와 xor 명령어 3회로 변경되었다고 볼 수 있다. 따라서 한 라운드마다 메모리 접근 명령어가 3회가 감소하였고 전체적으로 메모리 접근 회수가 30회 감소하여 속도향상에 기여한다. 복호화시에도 [그림 3]과 같이 복호화 부분 라운드키를 사용하여 고속구현이 가능하다.



[그림 2] 암호화시 부분 라운드키를 이용한 라운드키 생성 과정



[그림 3] 복호화시 부분 라운드키를 이용한 라운드키 생성 과정

4.2 실제 구현 기술과 결과

제안하는 EM64T 고속코드는 Subbytes, Shiftrows, Mixcolumns 부분은 bernstein(amd64-2)의 코드와 유사하게 설계하였으며, Addroundkey 부분은 부분 라운드키 기법으로 구현하였다. 치환 테이블은 2+2 KB를 사용하였는데, 이때 사용된 치환 테이블은 berstein이 사용한 2 KB 테이블과 마지막 라운드를 고속화하기 위한 2 KB 테이블을 추가하여 적용하였다. 라운드키를 할당한 4 개의 레지스터로 인해 레지스터가 부족했는데, 추가로 레지스터를 확보하기 위해 테이블 접근시 address 레지스터를 사용하지 않고 address 레이블을 통해 직접 접근하였다. [표 7]의 <코드 1>과 <코드 2>를 비교해보면, address 레지스터를 사용하면 address 레이블을 통해 접근하는 명령어보다 명령어 코드 크기가 짧아지는 장점이 있으나 EM64T에서는 속도 측면에서 차이가 없다.

부분 라운드키 기법을 적용하면 아래 수식 (2)와 같이 AES는 블록당 최소 182 회의 메모리 접근이 필요하다. 따라서 EM64T 프로세서에서는 반드시 블록당 182 사이클 이상 소요된다.

$$\begin{array}{l} \quad 4+4\text{회}(\text{평문읽기, 암호문쓰기}) \\ +\ 10\times 16\text{회}(\text{치환테이블참조}) \\ +\quad 14\text{회}(\text{부분라운드키읽기}) \\ \hline 182\text{ 회} \end{array} \qquad (2)$$

Matsui가 제시한 속도 측정방법[4]을 동일하게 적용하여 실험한 결과 185 cycles/block의 속도를 나타냈으며, Intel Core2Duo 3.0 Ghz 프로세서에서 ECB 모드로 2.0 Gbps의 처리속도를 보여주었다. 이 결과는 EM64T 프로세서에서 가장 빠르게 구현한 bernstein (amd64-2)보다 35 cycles/block 빠르게 개선되었다. 또한 수식 (2)와 같이 이론적인 한계속도는 182 cycles/block이며, 본 논문이 제시한 부분 라운드키 기법을 적용한 어셈블리 코드는 이론적인 한계점에 도달했다고 볼 수 있다. 부록 (1)은 부분 라운드키 기법을 적용한 실제 어셈블리 코드의 일부분으로 라운드 구현에 해당한다.

Ⅴ. AMD64 프로세서에서 고속 구현

AMD64 프로세서에서 이상적인 AES 고속 구현은 가장 적은 개수의 명령어를 최고의 효율로 동작시키는 것이다. 따라서 가장 먼저 고려해야 할 것은 명령어 개수를 최소화하는 것이고, 두 번째는 파이프라인의 병목현상을 해소하여 명령어 처리속도를 최대한 3 Instruction/cycle에 가까이 구현하는 것이다.

먼저 명령어 개수를 최소화하는 방법을 살펴보자.

bernstein(amd64-2) 코드가 라운드당 40 개의 명령어로 가장 적은 개수의 명령어로 구현하였다. 그러나 bernstein (amd64-2) 코드는 라운드의 워드단위 출력 A', B', C', D'의 값이 라운드함수의 마지막 부분에 생성되는 구조여서 메모리 접근 명령어로 이루어져 있는 Addroundkey가 라운드의 마지막에 치우치게 된다. 라운드의 뒤쪽으로 메모리 접근 명령어가 연속해서 배열되면 명령어 처리속도가 2 Instruction/cycle이 되므로 오히려 속도측면에서 불리하다. 반면 bernstein(amd64-1) 코드는 44개의 명령어로 구현되었지만 Addroundkey의 메모리 접근 명령어를 분산시킬 수 있어서 메모리 병목현상이 나타나지 않아 오히려 빠르다. 실험결과 라운드당 40 개의 명령어로 구현할 경우 메모리 접근 명령어가 뭉치는 구조가 발생하여 라운드당 44 개의 명령어로 구현된 bernstein(amd64-1) 코드보다 오히려 느려지는 현상이 나타났다. 제안하는 기법은 라운드당 44 개의 명령어를 사용하였다.

고속화 두 번째 단계로, 병목현상을 해소하여 3 Instruction/cycle에 가까이 구현하는 것이다. 앞서 언급한 바와 같이 AMD64 프로세서는 연속된 세 개의 명령어 코드의 크기가 16 바이트를 넘길 때, 한 사이클 이상이 소모된다. 따라서 연속된 세 개의 명령어 코드의 크기가 16 바이트를 넘지 않도록 명령어의 종류나 명령어의 배치를 조정해야 한다. 다음 [표 9]는 실제 코드로 활용되는 명령어들과 그 길이를 정리한 것이다. 같은 결과를 얻는 명령어라면 길이가 짧은 명령어를 사용하는 편이 속도향상에 유리하다.

[표 9] 64-비트 명령어의 코드 크기[14]

명령어	명령어 크기
xor eax, table0[r14*8]	8 바이트
xor eax, [r9+r14*8]	5 바이트
movzx esi, al	3 바이트
movzx rsi, al	4 바이트
shr eax, 16	3 바이트
shr rax, 16	4 바이트
mov eax, [r8+8]	4 바이트
mov eax, [r8+128]	7 바이트

실험을 통해 속도 향상에 유효했던 방법을 나열하면 다음과 같다. 첫째, 치환 테이블 참조할 경우 address 레지스터를 사용하여 명령어의 길이를 5 바이트로 줄인다. 둘째, 32-비트 명령어를 최대한 활용한다. 셋째, 라운드키 읽기에 사용되는 간접 address 상수는 0x80이 넘으면 명령어의 길이가 7 바이트로 증가한다. 따라서 간접 address 상수가 0x80에 가까워지면 라운드키 address 레지스터를 증가시켜주어 간접 address 상수가 0x80이 넘지 않도록 구현한다. 물론 address 레지스터를 증가시키기 위해 add 명령어를 추가해야 하지만 명령어 길이 감소에 따른 속도 향상 효과가 더 크게 나타났다. 넷째, 메모리 접근 명령어가 세 개 이상 연속해 있으면 파이프라인의 효율이 낮아지므로 메모리 접근 명령어 사이에 다른 명령어를 배치시킨다.

[표 10]은 AES 구현 소프트웨어 종류별 코드크기와 프로그램 속도의 관계를 보여준다. 본 논문에서 제안하는 AMD64 고속구현기법은 코드 크기가 168 bytes/round로 가장 작게 구현되었으며, 명령어 처리속도도 가장 빠르다. 제안하는 기법을 AMD Athlon64 프로세서에서 실험한 결과 170 cycles/block의 속도를 나타냈다. 이 결과는 Matsui가 구현한 비공개 코드의 결과와 동일한데, 이보다 빠르게 구현하지는 못했다. 부록 (2)에 제시된 코드는 AMD64 프로세서에 고도로 최적화한 AES 라운드 코드이다.

[표 10] AES 구현 소프트웨어의 명령어수, 코드크기와 AMD Athlon64에서 측정한 명령어 처리속도

소프트웨어	명령어수 (개/round)	평균 코드크기 (bytes/round)	명령어 처리속도 (Instruction/cycle)
Bernstein(amd64-1)	44	190	2.22
Bernstein(amd64-2)	40	185	2.02
EM64T 고속기법-부록 (1)	44	207	2.35
AMD64 고속기법-부록 (2)	44	168	2.75

참고로 앞에서 언급한 부분 라운드키를 이용한 EM64T 고속구현기법을 AMD64 프로세서에 적용해 본 결과 속도 향상효과를 보지 못했다. 가장 큰 이유는 AMD64 프로세서가 메모리 접근 명령어의 개수와 속도간의 상관성이 없기 때문이며, 부록 (1)에서 사용된 명령어들은 명령어 코드의 크기가 커서 라운드 코드 크기가 증가하였고 따라서 명령어 처리속도가 느려졌기 때문이다.

Ⅵ. 결 론

본 논문은 64-비트 프로세서인 EM64T 프로세서와 AMD64 프로세서에서 AES 알고리즘을 가장 빠르게 구현할 수 있는 구현 기법을 제시하였다. 본 논문의 부록에 첨부한 EM64T 고속구현 코드와 AMD64 고속구현 코드는 현재까지 가장 빠른 AES 구현 소프트웨어이며, 어떤 운영모드에서도 최고 성능을 나타내는 64-비트 프로세서용 AES primitive이다.

[표 11] 64비트 코드로 구현한 AES 구현 속도 비교

소프트웨어	Intel Core2Duo (cycles/block)	AMD Athlon64 (cycles/block)
Bernstein(amd64-1)	230	215
Bernstein(amd64-2)	220	218
Mitsuru Matsui[4]	251	170
EM64T 고속기법-부록 (1)	185	210
AMD64 고속기법-부록 (2)	226	170

이와 같은 성능을 구현하기 위해 EM64T 프로세서와 AMD64 프로세서에서 각각 다음과 같은 구현 기법을 제안하였다. 먼저 EM64T 프로세서에서는 한 사이클당 메모리 접근 명령어를 한 개뿐이 처리할 수 없기 때문에 메모리 접근 명령어의 비중이 높은 AES는 고속화에 불리하게 작용한다. 본 논문은 메모리 접근 명령어의 수를 줄이기 위해 라운드키를 메모리로부터 읽어오는 횟수를 44 회에서 14 회로 줄일 수 있는 부분 라운드키 기법을 제시하였다. 부분 라운드키 기법을 Intel Core2Duo 3.0 Ghz 프로세서에서 적용한 결과 ECB 모드에서 185 cycles/block, 2.0 Gbps의 성능을 보여주었다.

부분 라운드키 기법을 적용하면 수식 (2)와 같이 AES는 블록당 최소 182 회의 메모리 접근이 필요하다. 따라서 EM64T 프로세서에서는 반드시 블록당 182 사이클 이상 소요된다. 이론적인 한계속도는 182 cycles/block이며, 본 논문이 제시한 부분 라운드키 기법을 적용한 어셈블리 코드는 이론적인 한계점에 도달했다고 볼 수 있다.

한편 AMD64 프로세서에서는 패치 단계의 파이프라인에서 한 사이클에 단지 16 바이트의 명령어만 디코딩할 수 있기 때문에 길이가 긴 명령어의 조합을 수행할 때 성능저하가 나타났다. 따라서 명령어의 길이를 줄이는 기법에 초점을 맞추어 구현하였는데, AMD Athlon64 프로세서에서 구현한 결과 170 cycles/block의 속도에 도달하였다. 이 구현 결과는 Matsui의 비공개 코드의 측정결과와 동일하다. Matsui가 그의 논문[4]에서 언급한 바와 같이 AMD64의 latency를 고려할 때 160 cycles/block이 이론적인 한계라는 볼 수 있고, 두 결과는 이론적 한계에 근접하였다고 볼 수 있다.

참고문헌

[1] D. J. Bernstein, “AES speed”, 2008, available at: http://cr.yp.to/aes-speed.html

[2] Dag Arne Osvik, “Osvik Homepage”, 2005, available at: http://www.osvik.no

[3] Helger Lipmaa, “Fast Implementations”, available at: http://home.cyber.ee/helger/implementations

[4] Mitsuru Matsui, “How Far Can We Go on the 64-비트 Processors”, FSE 2006, LNCS 4047, pp. 341-358, 2006.

[5] Ralf S. Engelschall, “OpenSSL Project: The Open Source toolkit for SSL/TLS”, 2008, available at: http://www.openssl.org

[6] Mitsuru Matsui, Junko Nakajima, “On the Power of Bitslice Implementation on Intel Core2 Processor”, CHES 2007, LNCS 4727, pp. 121-134, 2007.

[7] Wu Hongjun, “Homepage of Hongjun Wu”, available at: http://icsd.i2r.a-star.edu.sg/staff/hongjun

[8] eSTREAM Project, “AES benchmarks : AES-CTR mode”, 2008, available at: http://www.ecrypt.u.org/stream/svn/iewcvs.cgi/crypt/trunk/benchmarks/aes-ctr/aes-128/?rev=216

[9] Brian Gladman, “Implementations of AES (Rijndael) in C/C++ and Assembler”, 2003, available at: http://fp.gladman.plus.com/cryptography_technology/rijndael

[10] Dag Arne Osvik, “Fast Assembler Implementations of the AES”. Presented at Crypto 2003 rump session, Santa Barbara, California, USA. Aug. 2003.

[11] Mitsuru Matsui, Sayaka Fukuda, “How to Maximize Software Performance of Symmetric

Primitives on Pentium III and 4 Processors", FSE 2005, LNCS 3557, pp. 398-412, 2005.

[12] Hans de Vries, "Understanding the detailed Architecture of AMD's 64 bit Core", Sep. 2003, available at: http://chip-architect.com/news/2003_09_21_Detailed_Architecture_of_AMDs_64bit_Core.html

[13] Michael Matz1, Jan Hubička, Andreas Jaeger, Mark Mitchell, "AMD64 ABI Draft 0.99", Dec. 2007, available at: http://www.x86-64.org/documentation/abi.pdf

[14] AMD, "Software Optimization Guide for AMD64 Processors", Sep. 2005, available at: http://amd.com/us-en/assets/content_type/white_papers_and_tech_docs/25112.pdf

[15] Torbjorn Granlund, Swox AB, "Instruction latencies and throughput for AMD and Intel x86 processors", Nov. 2007, available at: http://www.swox.com/doc/x86-timing.pdf

[16] NIST, "Federal Information Processing Standards Publication 197, Advanced Encryption Standard (AES)", 2001.

[17] Joan Daemen, Vincent Rijmen, "AES Proposal : Rijndael", 1998.

[18] Anger Fog, "Instruction tables", 2008.

<著 者 紹 介>

정 창 호 (Changho Jung) 정회원
2002년 2월 : 고려대학교 응용생명환경화학과 학사
2004년 8월 : 고려대학교 정보보호대학원 석사
2005년 3월～현재 : 한국전자통신연구원 부설연구소 연구원
<관심분야> 정보보호, 암호분석, 컴퓨터 포렌식

박 일 환 (IlHwan Park) 정회원
1988년 2월 : 고려대학교 수학과 졸업
1990년 2월 : 고려대학교 수학과 석사
1996년 2월 : 고려대학교 수학과 박사
1996년 5월～1999년 12월 : 한국전자통신연구원
2000년 1월～현재 : 한국전자통신연구원 부설연구소 책임연구원
<관심분야> 정보보호이론

부 록

(1) EM64T 프로세서에 고속 구현한 AES 라운드 코드

```
    .MACRO nround   rkA
           movzbl   %dl,%esi                    ;D0
           movzbl   %dh,%edi                    ;D1
           shr      $16,%edx
           movzbl   %dl,%r13d                   ;D2
           movzbl   %dh,%ebp                    ;D3
           movl     table0(,%rsi,8),%edx        ;D'=T0[D0]
           movzbl   %al,%r14d                   ;A0
           movzbl   %ah,%esi                    ;A1
           xorl     table1(,%rsi,8),%edx        ;D'^=T1[A1]
           shr      $16,%eax
           movzbl   %al,%r15d                   ;A2
           movzbl   %ah,%esi                    ;A3
           movl     table3(,%rbp,8),%eax        ;A'=T3[D3]
           movq     \rkA(%r8),%r9               ;rkA
           xorl     table0(,%r14,8),%eax        ;A'^=T0[A0]
           movzbl   %bl,%r14d                   ;B0
           movzbl   %bh,%ebp                    ;B1
           xorl     table1(,%rbp,8),%eax        ;A'^=T1[B1]
           shr      $16,%ebx
           movzbl   %bl,%ebp                    ;B2
           xorl     table2(,%rbp,8),%edx        ;D'^=T2[B2]
           movzbl   %bh,%ebp                    ;B3
           movl     table2(,%r13,8),%ebx        ;B'=T2[D2]
           xorq     %r9,%r10                    ;rkB
           xorl     table3(,%rsi,8),%ebx        ;B'^=T3[A3]
           xorq     %r10,%r11                   ;rkC
           xorl     table0(,%r14,8),%ebx        ;B'^=T0[B0]
           movzbl   %cl,%r13d                   ;C0
           movzbl   %ch,%esi                    ;C1
           xorl     table1(,%rsi,8),%ebx        ;B'^=T1[C1]
           shr      $16,%ecx
           movzbl   %cl,%esi                    ;C2
           xorl     table2(,%rsi,8),%eax        ;A'^=T2[C2]
           movzbl   %ch,%ecx                    ;C3
           xorl     table3(,%rcx,8),%edx        ;D'^=T3[C3]
           xorq     %r11,%r12                   ;rkD
           movl     table1(,%rdi,8),%ecx        ;C'=T1[D1]
           xorq     %r9,%rax                    ;A'^=rkA
           xorl     table2(,%r15,8),%ecx        ;C'^=T2[A2]
           xorq     %r10,%rbx                   ;B'^=rkB
           xorl     table3(,%rbp,8),%ecx        ;C'^=T3[B3]
           xorq     %r11,%rcx                   ;C'^=rkC
           xorl     table0(,%r13,8),%ecx        ;C'^=T0[C0]
           xorq     %r12,%rdx                   ;D'^=rkD
    .ENDM
```

(2) AMD64 프로세서에 고속 구현한 AES 라운드 코드

```
.MACRO nround   rkA,rkB,rkC,rkD
        movzbl  %al,%edi                    ;A0
        movl    table0(%r9,%rdi,8),%r10d    ;A'=T0[A0]
        movzbl  %bl,%edi                    ;B0
        movl    table0(%r9,%rdi,8),%r11d    ;B'=T0[B0]
        movzbl  %cl,%edi                    ;C0
        movl    table0(%r9,%rdi,8),%r12d    ;C'=T0[C0]
        movzbl  %dl,%edi                    ;D0
        movl    table0(%r9,%rdi,8),%r13d    ;D'=T0[D0]
        movzbl  %ah,%edi                    ;A1
        xorl    table1(%r9,%rdi,8),%r13d    ;D'^=T1[A1]
        movzbl  %bh,%edi                    ;B1
        xorl    table1(%r9,%rdi,8),%r10d    ;A'^=T1[B1]
        movzbl  %ch,%edi                    ;C1
        xorl    table1(%r9,%rdi,8),%r11d    ;B'^=T1[C1]
        movzbl  %dh,%edi                    ;D1
        xorl    table1(%r9,%rdi,8),%r12d    ;C'^=T1[D1]
        shr     $16,%edx
        shr     $16,%ebx
        shr     $16,%ecx
        shr     $16,%eax
        movzbl  %al,%edi                    ;A2
        xorl    table2(%r9,%rdi,8),%r12d    ;C'^=T2[A2]
        movzbl  %bl,%edi                    ;B2
        xorl    table2(%r9,%rdi,8),%r13d    ;D'^=T2[B2]
        movzbl  %cl,%edi                    ;C2
        xorl    table2(%r9,%rdi,8),%r10d    ;A'^=T2[C2]
        movzbl  %dl,%edi                    ;D2
        xorl    table2(%r9,%rdi,8),%r11d    ;B'^=T2[D2]
        movzbl  %ah,%edi                    ;A3
        xorl    table3(%r9,%rdi,8),%r11d    ;B'^=T3[A3]
        movzbl  %bh,%edi                    ;B3
        xorl    table3(%r9,%rdi,8),%r12d    ;C'^=T3[B3]
        movzbl  %ch,%edi                    ;C3
        xorl    table3(%r9,%rdi,8),%r13d    ;D'^=T3[C3]
        movzbl  %dh,%edi                    ;D3
        xorl    table3(%r9,%rdi,8),%r10d    ;A'^=T3[D3]
        movl    \rkA(%r8),%eax              ;rkA
        movl    \rkB(%r8),%ebx              ;rkB
        xorq    %r10,%rax                   ;A'^=rkA
        xorq    %r11,%rbx                   ;B'^=rkB
        movl    \rkC(%r8),%ecx              ;rkC
        movl    \rkD(%r8),%edx              ;rkD
        xorq    %r12,%rcx                   ;C'^=rkC
        xorq    %r13,%rdx                   ;D'^=rkD
.ENDM
```