논문 12-37C-10-10

한국통신학회논문지 '12-10 Vol.37C No.10
http://dx.doi.org/10.7840/kics.2012.37C.10.940

안드로이드 기반 공개키 암호를 위한 곱셈기 구현 및 분석

서 화 정[•], 김 호 원[°]

Implementation and Analysis of Multi-Precision Multiplication for Public Key Cryptography Based on Android Platform

Hwa-jeong Seo[•], Ho-won Kim[°]

요 약

안드로이드 프로그램은 JAVA SDK로 제작되어 가상머신(virtual machine) 기반으로 동작한다. 따라서 기존의 C 언어에 비해 프로그램 작성은 편리해 졌지만 동작 속도는 떨어지는 단점이 있다. 이러한 단점을 보완하기 위해 안드로이드 상에서 C언어, 어셈블리 언어의 사용이 가능한 안드로이드 NDK가 제안되어 보다 효율적인 프로그램 작성이 가능하게 되었다. 이와 더불어 ARM에서 제공하는 NEON기능을 사용하면 벡터연산을 통해 성능을 향상 시킬 수 있다. 본 논문에서는 NDK의 효용성에 대해 알아보며 NEON기능을 이용한 향상된 곱셈구조를 제안한다.

Key Words : NDK, NEON, ANDROID, ARM, Multiplication

ABSTRACT

Android program is developed with JAVA SDK and executed over virtual machine. For this reason, programming is easier than traditional C language but performance of operating speed decreases. To enhance the performance, NDK development tool, which provides C language, assembly language environment, was proposed. Furthermore, with NEON function provided by ARM, we can utilize the vector operation and enhance performance. In the paper, we explore effectiveness of NDK and then propose advanced multiplication structure with NEON function.

Ⅰ. 서 론

구글에서 개발한 안드로이드 플랫폼은 높은 완성도와 공개성으로 인해 스마트 폰에서의 운영체제로 가장 널리 사용되는 플랫폼이다[4,14]. 현재 안드로이드는 지속적으로 새로운 버전을 릴리즈하며 기능과 안정성을 높여가고 있는 추세이다. 안드로이드는 앱의 형태로 사용자에게 제공되며 이는 게임에서부터 시작하여 인터넷 뱅킹에 이르기까지 우리 삶 전반에 걸쳐 모든 분야에 적용되어 가고 있는 상황이다. 이와 더불어 무선을 통해 통신하는 개인의 정보 역시 이전에 비해 보다 쉽게 노출되고 있다. 이를 해결하기 위해서는 스마트 폰 상에서 전송되는 정보들이 암호화되어 안전하게 전달되도록 해야 한다.

기본적으로 스마트폰 상에서는 API로 제공하는 다양한 대칭키, 공개키 그리고 해시모듈을 통해 사용자가 전송하는 정보를 안전하게 보호하는 것이 가능하다. 하지만 API는 JAVA 상에서 패키지 형식으로 제공되므로 하위 단의 언어, 즉 C언어와 어셈블리(assembly)언어로 작성될 때보다는 성능이 떨어지게 된다. 스마트폰과 같은 자원한정적인 임베디드 장비에서는 최적화된 암호화모듈 구현을 통해 암호화과정을

※ 이 논문은 2010년도 정부(교육과학기술부)의 재원으로 한국연구재단의 지원을 받아 수행된 연구임(No.2010-0026621).
◆ 주저자 : 부산대학교 컴퓨터공학과 정보보호 연구실, hwajeong@pusan.ac.kr, 준회원
° 교신저자 : howonkim@pusan.ac.kr, 종신회원
논문번호 : KICS2012-06-278, 접수일자 : 2012년 6월 18일, 최종논문접수일자 : 2012년 10월 8일

단축시키고 스마트폰을 최대한 절전모드로 유지하는 것이 중요하다. 기존의 안드로이드 SDK에서는 JAVA를 통한 구현만이 가능했다[5]. 하지만 보다 탄력적인 개발환경을 위해 2009년 6월에 안드로이드 NDK가 구글에 의해 공개되었다[6,15]. 이는 장비를 조작하기 위한 프레임워크(framework)단의 한계를 넘어 C언어로 안드로이드를 동작시키는 것이 가능하게 하여 기존의 자바기반의 프로그램 구현에 비해 속도가 개선되는 효율성을 가진다. 따라서 사용자가 이를 효과적으로 사용한다면 프로그램의 성능 향상이 가능하다. 하지만 C언어 역시 비트연산과 같은 부분에서는 최적화된 연산이 불가능하다. 따라서 이를 개선하기 위해서는 보다 하위단의 언어인 어셈블리(assembly)언어를 통한 개발이 요구된다. 이와 더불어 최근에는 ARM 프로세서 NEON 기능을 제공하는 A8, A9 프로세서를 출시하여 벡터연산을 효율적으로 수행하는 것이 가능하도록 하였다. NEON을 사용하면 보다 여러 개의 동일한 연산을 한 번에 수행하는 것이 가능하다.

본 논문에서는 NDK를 통한 공개키 기반 암호화를 위한 곱셈(multiplication)기를 어셈블리(assembly)를 통해 구현한다. 해당 어셈블리(assembly)는 NEON기능을 통해 벡터 연산이 가능한 형태로 구성되어 제안된다. 본 논문의 구성은 다음과 같다. 2장에서는 현재 안드로이드의 전체적인 동작방식과 NDK를 통한 프로그래밍에 대해 살펴본다. 3장에서는 본 논문에서 기존의 곱셈(multiplication) 구현기법을 소개하며 NDK를 사용한 구현의 성능향상을 확인해 본다. 4장에서는 NEON기능을 통한 구현기법을 제시하며. 5장에서는 제안된 기법의 성능을 분석하며 마지막으로 6장에서는 본 논문에 대한 결론을 내린다.

Ⅱ. 논문 인용의 예

2.1. JNI[1.9]

안드로이드 프레임 워크에서 C/C++과 자바와 유기적으로 연결되기 위해서는 상위계층인 자바와 하위계층인 C/C++레이어를 상호 연결해 주는 매개체가 필요하다[7,8]. [그림 1]과 같이 안드로이드에서 자바와 C/C++ 모듈간의 인터페이스가 가능하게 해주는 것은 JNI(Java Native Interface)이다. JNI는 다음과 같은 기능을 활용하기 위해 많이 사용되고 있다.

◇ 빠른 처리속도를 요하는 프로그램: JNI를 통한 C/C++언어를 통한 프로그램 작성은 자바를 통한 프로그램 작성에 비해 최적화된 코드를 제공함으로써 일반적으로 자바에 비해 높은 성능을 보인다. 따라서 성능이 중요한 프로그램의 경우 JNI를 통해 C/C++을 직접 동작시킬 수 있도록 한다.

◇ 하드웨어 제어: 하드웨어 제어 코드는 C언어상에서 작성이 가능한데 이를 JNI를 통해 자바에서도 조작이 가능하게 된다.

◇ 프로그램 재사용: 기존에 C/C++로 제작되어 사용되고 있는 라이브러리와 프로그램의 경우 자바에서 사용하기 위해서는 JNI를 통해 C/C++에 대한 인터페이스를 제공받아 활용하는 것이 가능하다.

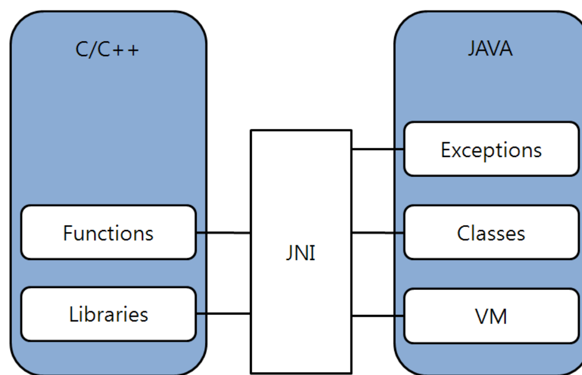

그림 1. 자바와 C언어를 연결하는 매개체
Fig. 1 Medium for connecting between JAVA and C language

2.2 NDK와 SDK에 대한 비교

안드로이드 상에서의 프로그램의 구현은 현재 자바를 이용하여 달빗 가상머신 상에서 구현된다[6]. 하지만 성능상의 이유로 기존의 SDK방식이 아닌 NDK를 이용한 프로그램 작성이 본격화되고 있다. NDK를 활용한 구현 기법은 물리 시뮬레이션과 신호처리와 같은 영역에 보다 효과적이다. 또한 기본에 작성된 C/C++언어를 재사용하는 것이 가능하다는 장점을 가진다. 기본적인 프로그램 작성 시 NDK를 통한 프로그램의 성능은 SDK를 사용할 때에 비해 10배~20배 정도의 향상을 보인다. 하지만 NDK를 통한 프로그램 작성 시 C언어를 통해 작성된 부분을 호출하기 위해서는 문맥교환(context switching)시 많은 부하가 생기므로 호출 횟수를 최적화하고 한 번 호출 시 많은 연산을 수행하도록 작성되어야 한다[19].

2.3 Android BigInteger API[2]

안드로이드에서는 기본적으로 java.math.* 하위 라이브러리로 Big Integer구현에 대한 패키지를 포함한다. 따라서 사용자는 쉽게 해당 라이브러리를 포함하

여 큰 비트에 대한 사칙연산이 쉽게 가능하다. 본 논문에서는 성능을 비교하기 위해 BigInteger API상에서 임의의 BigInteger 두 개를 생성하고 이를 곱하는 연산을 수행한다.

2.5 이전연구 결과

2.5.1. Leonid Batyuk et al[11]

해당 논문에서는 안드로이드 환경과 자바 환경과의 성능의 차이점을 JRE 1.6을 이용하여 평가하였다. 정렬(sorting)알고리즘에 대한 성능을 JNI와 기본 코드(native code)와의 조합을 통해 나타내었다. 안드로이드 환경 상에서는 JNI와 기본 코드(native code)를 사용하는 것이 Dalvik 가상머신(virtual machine)만을 사용하는 것보다 성능이 좋게 나왔다. 하지만 Sun JRE의 경우에는 JNI와 기본 코드(native code)를 함께 쓰는 것이 JVM만을 사용하는 것보다 성능이 좋게 나오지는 않았다. 이는 개발 툴에 따라 최적화되는 기법과 옵션 때문이다.

2.5.2. Lee, Jeon et al[12]

해당 논문에서는 JNI를 사용할 때 발생하는 지연, 정수 계산, 실수 계산, 메모리 접근 알고리즘 그리고 힙 메모리 할당(heap memory allocation) 알고리즘에 대한 실험을 수행하였다. 실험 결과 JNI를 사용할 때 발생하는 지연은 다른 연산들의 계산 시 발생하는 이득으로 인해 상쇄되며 나머지 실험에서도 모두 NDK를 사용하여 구현한 결과가 성능이 높게 평가되었다. 특히 메모리와 heap에 대한 성능에서는 매우 큰 성능의 차이를 보였다. 하지만 JNI지연은 상당히 큰 부하를 요구하므로 가능한 JNI 호출 횟수를 줄이는 것이 효율적이다.

2.5.3. Paik, Lee et al[13]

해당 논문에서는 JAVA를 통해 구현한 암호화 기법과 NDK를 사용했을 때의 성능을 비교 평가하고 있다. 구현 시 최적화 알고리즘이 적용되지 않았지만 JAVA 상에서 제공하는 API에 비해 높은 성능을 낸다는 것은 NDK가 성능이 중요시 되는 프로그램에서는 꼭 사용되어야 함을 나타낸다. 또한 논문에서는 NDK를 보다 효율적으로 사용하기 위한 프레임워크(framework)을 제안하여 보다 효율적으로 NDK를 활용할 수 있도록 하고 있다.

Ⅲ. 안드로이드 상에서의 Big Integer 연산 모듈 구현

곱셈 연산은 공개키 기반 암호화에 요구되는 큰 비트 상에서의 사칙연산으로써 임베디드 시스템은 비트의 크기가 8, 16, 32비트로 정해지므로 큰 비트에 대한 사칙연산의 구현이 어렵다. 지금까지 자원 한정적인 디바이스 상에서의 다양한 곱셈(multiplication)기법이 제안되어 메모리에 대한 접근을 최소화하는 방안이 제시되어 왔다. 본 장에서는 공개키 기반 암호화의 기본 연산인 Big Integer연산을 보다 효율적으로 안드로이드 상에서 구현하기 위한 기법을 확인한다. 이를 위해 기본적인 곱셈(multiplication)기법을 JAVA 상에서의 BigInteger API, NDK을 통한 C언어와 어셈블리(assembly) 언어 구현 성능을 확인한다. 그리고 기본적인 ARM의 구조에 적합한 곱셈기법의 성능을 시뮬레이터를 통해 확인한다.

3.1. 곱셈구현기법

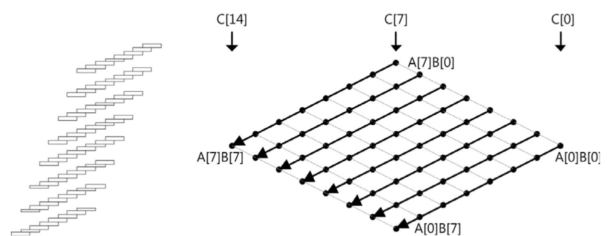

그림 2 Row-wise 곱셈기
Fig. 2. Row-wise multiplication

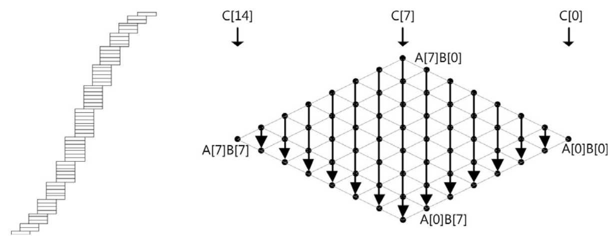

그림 3. Column-wise 곱셈기
Fig. 3. Column-wise multiplication

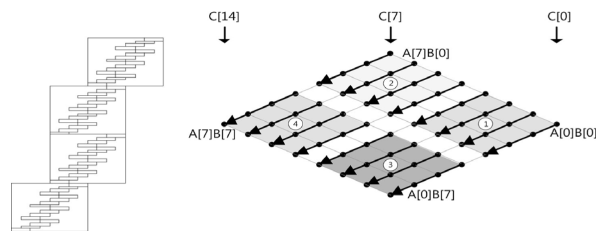

그림 4. Hybrid 곱셈기
Fig. 4. Hybrid multiplication

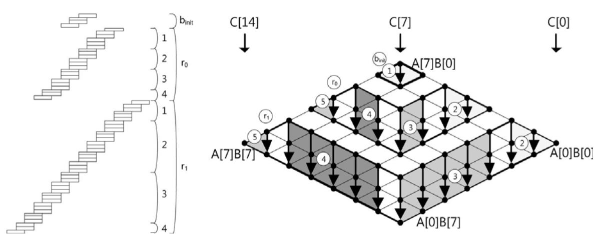

그림 5. Operand Caching 곱셈기
Fig. 5. Operand Caching multiplication

Row-wise 곱셈기는 같은 행에 위치한 모든 인자값을 한 번에 읽어와서 곱셈을 취하는 방식이다. 해당 방식은 하나의 인자값에 대한 모든 곱셈 결과를 얻을 수 있다는 장점이 있지만 지속적으로 중간 결과값과 인자값을 불러와야 하는 단점을 가진다. Row-wise 곱셈기에 대한 예시는 [그림 2]에 자세히 나타나 있다. [그림 2]에서는 총 8개의 A, B인자에 대한 곱셈을 통해 15개의 C결과값을 도출해 내는 연산을 수행한다. A[0]이 인자로 불러오게 되면 B의 경우 0~7에 해당하는 모든 인자값을 불러와 A[0]과의 부분 곱셈값을 도출하게 된다. 해당 결과값은 C의 0~8에 해당하는 메모리 주소에 값을 저장하게 된다. 그 다음 연산은 A[1]에 대한 부분 곱셈을 수행하게 되며 지속적으로 모든 A의 인자에 대한 곱셈 연산을 수행하게 된다. 따라서 B에 대한 인자를 모두 저장하기 위한 레지스터와 결과값을 저장하기 위한 레지스터를 모두 확보해야하는 구조를 가진다. 만약 충분한 레지스터가 보장이 된다면 매우 효과적으로 연산이 가능하지만 임베디드 장비는 한정적인 레지스터로 인해 효율적인 연산을 기대하기 힘들다. 따라서 Row-wise 곱셈기는 임베디드 장비 상에서는 효율적인 곱셈 기법이 아니다.

Column-wise 곱셈기는 같은 열에 위치한 부분곱셈을 한 번에 모두 수행하는 방식으로써 중간 결과값을 다시 읽어오지 않아도 되는 장점을 가진다[3]. 해당 기법은 동일한 열에 위치한 인자들을 다 읽어와 연산을 수행하고 해당 열의 모든 부분 곱셈을 수행하고 난 다음에는 다시 돌아와서 해당 열을 계산할 필요가 없으므로 지속적인 메모리 접근을 효율적으로 줄일 수 있다.

[그림 3]에서 나타난 Column-wise 곱셈기 기법에서는 화살표 방향을 보면 동일한 결과값 C의 인자에 대한 부분 곱셈을 한 번에 다 수행함을 확인할 수 있다. 이는 인자를 불러오기 위한 연산의 증가를 불러오지만 축적해야하는 레지스터의 크기가 곱셈결과값보다 하나가 크기 때문에 레지스터의 수가 적은 임베디드 장비상에서는 보다 효율적으로 연산이 가능하다. C[7]열을 예시로 들어보면 A[7]과 B[0] 그리고 A[6]과 B[1]과 같이 동일한 열에 대한 연산이 수행되므로 결과값의 축적이 매우 용이하다. [그림 4]에 나타난 Hybrid 곱셈기는 기존 Row-wise 곱셈기와 Column-wise 곱셈기 각각의 장점인 한 번에 곱셈에 필요한 모든 인자들을 불러와서 곱셈하는 기법과 결과값을 축적하는 레지스터의 개수를 최소화하는 방안을 혼합하여 사용하는 기법이다. 전체 곱셈은 큰 블락 단위로 묶어서 나타낼 수 있다. 해당 블락은 Column-wise형식으로 연산이 수행되며 블락은 다시 여러번의 Row-wise곱셈이 내부에서 수행되는 방식으로 설계되어 있다 해당 기법은 레지스터에 대한 효율성을 극대화함으로써 메모리에 대한 접근을 줄이게 된다[16].

Hybrid 곱셈에 대한 예시는 [그림 4]에 나타나 있다. 큰 블락은 1, 2, 3 그리고 4와 같은 형식으로 총 16번의 부분곱셈을 수행하며 이는 하나의 블락으로 묶어서 표현한다. 곱셈은 번호 순으로 1, 2, 3 그리고 4의 차례로 수행되며 해당 블락의 내부에서는 Row-wise 곱셈방식으로 곱셈이 수행된다. [그림 5]에 나타난 Operand Caching 기법은 기존의 기법들이 최대한 중간 결과값에 대한 접근을 피했다면 해당 기법은 중간 결과값에 대한 접근을 인자들에 대한 접근과 효율적으로 대치시키며 성능을 향상시켰다[17]. 만약 곱셈에 수행되어야 하는 인자가 각각 10개씩 존재한다면 부분 곱셈을 10개의 쌍에 대해 취한 결과값을 구하는 것이 가능하다. 그 다음에 결과값을 구하기 위해 인자를 선택해야 하는 시점이 오면 이전에 사용했던 인자의 쌍 중 하나의 인자들의 집합을 다음 연산 결과에도 사용하여 인자에 대한 접근을 효율적으로 줄일 수 있는 구조로 되어 있다.

Operand Caching에 대한 예시는 [그림 5]과 같다. 먼저 가장 상위단에 위치하는 b_{init}을 계산한 후 다음 열에 해당하는 r_1을 계산한다. 이때 그림에서와 같이 2, 3, 4 그리고 5의 순서로 부분곱셈이 수행되게 된다. 여기서 2와 3은 A인자들의 집합을 공유하게 되며 3, 4 그리고 5의 연산에서는 B인자들의 집합을 공유하게 된다. 따라서 지속적으로 동일한 인자를 유지하며 사용하게 되므로 인자를 불러오기 위한 메모리 접근을 효율적으로 줄일 수 있다.

3.2. 안드로이드 곱셈구현 기법

3.2.1. JAVA의 BigInteger API를 통한 곱셈구현

안드로이드 API에서는 BigInteger 패키지가 있으며 해당 패키지를 통해 사용자는 안드로이드 상에서 쉽게 큰 인수 연산의 수행이 가능하다.

3.2.2. C언어를 통한 곱셈구현

C언어를 통한 곱셈(multiplication)의 구현에 사용한 곱셈(multiplication) 기법은 Product Scanning 곱셈(multiplication)기법으로써 동일한 열에 위치한 값들을 지속적으로 축적하며 마지막에 한번만 해당 결과 값을 저장(store)하는 방식이다. 따라서 중간 값을 저장(store)하기 위해 메모리에 대한 접근하는 횟수가 줄어드는 장점을 가진다. 곱셈(multiplication) 구현 시 고려해야 하는 부분은 정의하는 변수의 형식이다. 안드로이드에서 제공하는 변수의 형식은 [표 1]과 같으며 32-비트 이상에서는 unsigned 형식을 지원하지 않는다. C언어를 통한 곱셈(multiplication) 구현 시 32-비트 덧셈(addition)을 하게 되는 경우 올림이 발생하는 경우가 있는데 이때 추가적인 연산을 통해 올림을 확인하도록 한다. 해당 함수는 JNI를 통해 자바의 함수와 C언어의 곱셈(multiplication)함수와 연결되어 유기적으로 곱셈(multiplication)명령을 수행하게 된다. 즉 자바코드로 작성된 부분이 수행되게 되면 JNI를 통해 이와 연관된 C언어 구현로 구현된 곱셈(multiplication)을 수행하게 된다. 곱셈(multiplication)의 구현은 Column-wise기법에 따라 동일한 열에 위치한 값을 축적시키며 연산을 하도록 구현하였다.

표 1 JNI상에서의 변수 형식
Table 1. JNI variable type

C Type	JAVA Type	Description
boolean	jboolean	unsigned 8-bit
byte	jbyte	signed 8-bit
char	jchar	unsigned 16-bit
short	jshort	signed 16-bit
int	jint	signed 32-bit
long	jlong	signed 64-bit
float	jfloat	32-bit
double	jdouble	64-bit

3.2.3. 어셈블리(assembly) 언어를 통한 곱셈구현

안드로이드 상에서의 어셈블리(assembly) 코딩은 ARM에서 제공하는 기본적인 어셈블리(assembly) 코딩이 가능하도록 되어 있다. 곱셈은 동일하게 column-wise기법으로 수행되며 해당 열에 대한 곱셈(multiplication)이 끝나면 결과값은 메모리에 저장(store)되게 된다. 어셈블리(assembly) 프로그램에서는 중간 결과값을 축적하는 상황에서 발생하는 올림값을 현재 결과값과 더한 값을 연산할 수 있도록 되어 있다. 따라서 C언어를 통한 연산에 비해 최적화된 구현이 가능하다.

3.3. NDK를 이용하여 구현한 곱셈(multiplication) 모듈의 성능 분석

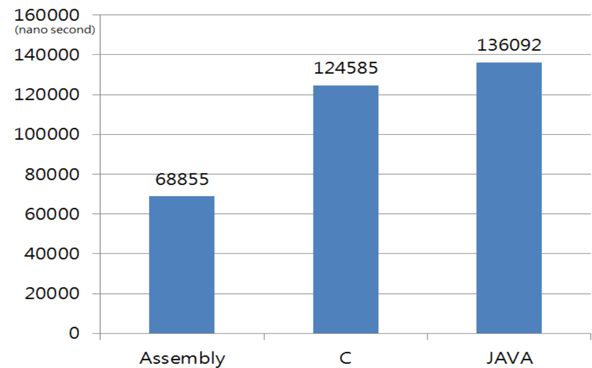

그림 6. 구현 결과 속도 비교
Fig. 6. Comparison of performance

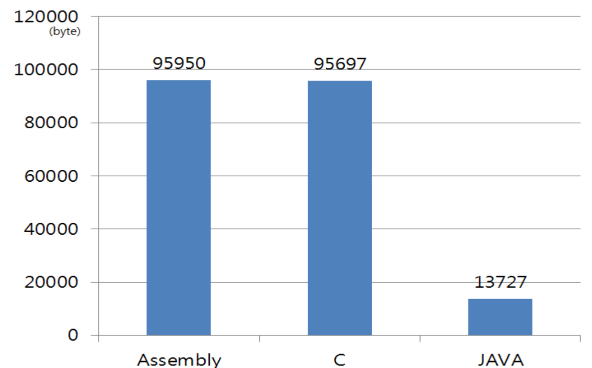

그림 7. 구현 결과 memory 비교
Fig. 7 Comparison of memory consumption

본 장에서는 기존의 API와 NDK를 사용한 최적화 곱셈(multiplication) 구현의 성능을 확인해보도록 한다. 그 결과는 [그림 6]과 같다. 그림에서 보는바와 같이 현재 안드로이드에서 제공하는 API는 NDK에 비해 성능이 저하됨을 나타낸다. 그 이유는 자바를 통한 구현에서는 가상머신(virtual machine) 위에서 함수 호출이 이루어지기 때문에 최적화보다는 다양한 플랫폼 상에서 수행이 가능하도록 작성되어 있다. 하지만 NDK의 경우에는 안드로이드 플랫폼 상에서 JAVA 단에서의 프로그래밍이 아닌 하위단의 C언어와 어셈블리(assembly)를 통해 구현되

므로 보다 최적화된 구현이 가능하다. 성능은 하위단의 언어인 어셈블리(assembly)를 통해 구현 시 가장 높게 측정되었으며 C언어의 경우 JAVA에 비해 약간 향상된 성능을 제시하였다.

속도와 함께 고려해야 할 부분은 바로 사용되는 메모리양이다. [그림 7]은 C언어와 자바 구현 시 생성되는 apk파일의 크기를 비교한 결과를 나타낸다. [그림 7]에서 자바 API를 통해 생성된 파일은 13.4 KB의 용량을 나타낸다. C언어를 통해 생성된 파일은 93.5KB를 나타낸다. 마지막으로 어셈블리(assembly)를 통해 생성된 파일은 93.7KB를 나타낸다. 즉 구현 속도를 향상시키기 위해 하위단의 언어를 사용하여 최적화된 코드로 작성하게 되면 속도는 향상되지만 그만큼 코드의 양이 많아지기 때문에 패키지의 크기가 증가하는 단점을 가진다. 반면에 자바로 작성된 코드는 고급언어로 축약된 형태로 명령이 수행되기 때문에 작은 용량을 나타낸다. 결론적으로 속도와 메모리는 서로간의 반비례관계에 있으므로 응용에 따라 성능을 조절하는 것이 중요하다.

Ⅳ. 제안하는 알고리즘

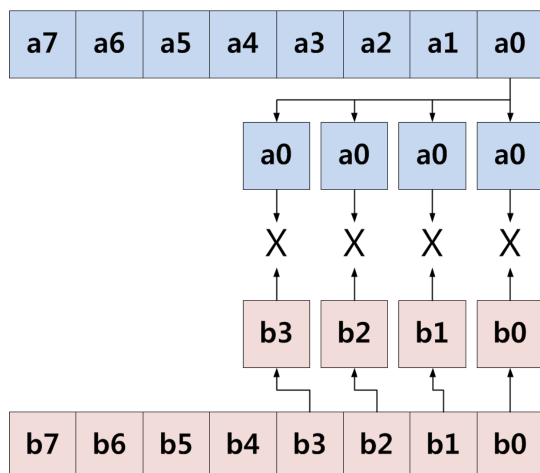

그림 8. 제안하는 곱셈기법
Fig. 8. Proposed multiplication

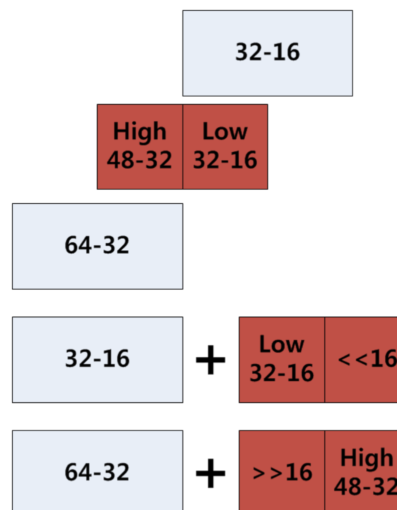

그림 9. 중첩영역 제거
Fig. 9 Elimination of duplicated area

ARMv7는 SIMD(Single Instruction Multiple Data)를 통해 벡터연산에 효율적인 NEON기능을 제공하며 이는 효과적으로 많은 데이터를 한 번의 연산으로 계산하는 것이 가능하다. 따라서 연산되는 데이터 간에 종속성이 존재하지 않는 경우 병렬적으로 동시에 여러 연산이 가능하게 되어 효과적이다. 본 논문에서는 최근에 가속화되고 있는 NEON 기능을 이용한 새로운 곱셈기법을 제안한다. 본 논문에서 제안한 기법은 Row-wise 곱셈 기법을 NEON의 벡터기능과 결합하여 사용하였다. 이에 대한 설명은 [그림 8]과 같다. 두 개의 인자 a와 b에 대한 곱셈을 계산하는 경우 먼저 a의 인자 중 첫 번째 인자인 a0을 벡터의 모든 값에 인가하도록 한다. 그 다음 b의 인자들을 순서대로 b0,b1,b2,b3를 인가하여 이에 대한 곱셈을 구하도록 한다. 해당 곱셈연산이 수행되고 나면 b의 나머지 인자값 b4, b5, b6, b7과 a0의 곱셈을 수행하게 된다. b에 대한 모든 곱셈값이 구해지고 나면 a의 인자를 하나 증가시킨 후 해당 인자에 대한 모든 곱셈값을 구하게 된다. 곱셈이 수행되는 순서에 관해서는 표2와 같이 나타낼 수 있다. 해당 예시는 160비트 곱셈을 16비트 곱셈으로 수행할 때의 벡터곱셈의 순서와 벡터의 구성을 나타낸다. 벡터를 구성할 때 64비트씩 벡터를 구성하게 되는 경우 160비트는 64비트의 배수가 되지 않으므로 32비트가 불필요한 연산이 수행된다. 따라서 곱셈의 파라미터가 64비트의 배수가 되는 경우 벡터의 Row-wise곱셈기법의 성능이 가장 좋게 나온다고 할 수 있다.

제안하는 곱셈기법의 장점은 한번에 4번의 곱셈을 수행하여 성능이 향상되며, a인자를 동일한 값으로 사

용하기 때문에 곱셈 시 레지스터에 값을 추가적으로 갱신할 필요없이 모든 벡터의 계산이 가능하다. 또한 곱셈의 결과값이 비트의 오름차순으로 발생하기 때문에 결과값을 저장하고 불러오는 데 용이하다. 예를 들어 만약 16비트 벡터 곱셈을 취했을 때 a0과 b0,b1,b2,b3값을 곱하게 되면 결과값은 각각 0~32, 16~48, 32~64, 48~80 비트 자리에 결과값을 출력하게 된다.

하지만 여기서 문제가 발생하게 되는 부분은 곱셈을 취하는 값을 순차적으로 하게 될 경우 16~32, 32~48, 48~64 비트와 같이 중첩되는 경우가 발생하게 된다. 이를 해결하기 위해 본 논문에서는 [그림 9]에서와 같이 ARM의 덧셈연산이 시프트 연산을 포함할 수 있다는 점에 착안하여 중간에 중첩되는 값을 16비트씩 시프트한 값과 덧셈을 취하여 해당 문제를 해결하도록 하였다. 덧셈을 통해 중첩된 부분을 제거하는 경우 하위 16비트의 덧셈값은 올림이 발생하지만 상위 16비트의 값은 어떤값이 들어가더라도 올림이 발생하지 않는다. 그 이유는 16비트 곱셈의 결과 발생하는 최대 결과값은 0xFFFF X 0xFFFF = 0xFFFE0001 이다. 따라서 하위 비트에 의해 올림이 발생하는 경우에도 상위 비트의 값에는 영향을 주지 않는 특징을 가진다.

표 2. 제안된 기법의 곱셈 순서
Table 2. Multiplication Sequence of Proposed Method

(a0 x b0, a0 x b1, a0 x b2, a0 x b3), (a0 x b4, a0 x b5, a0 x b6, a0 x b7), (a0 x b8, a0 x b9, 0, 0)
(a1 x b0, a1 x b1, a1 x b2, a1 x b3), (a1 x b4, a1 x b5, a1 x b6, a1 x b7), (a1 x b8, a1 x b9, 0, 0)
(a2 x b0, a2 x b1, a2 x b2, a2 x b3), (a2 x b4, a2 x b5, a2 x b6, a2 x b7), (a2 x b8, a2 x b9, 0, 0)
(a3 x b0, a3 x b1, a3 x b2, a3 x b3), (a3 x b4, a3 x b5, a3 x b6, a3 x b7), (a3 x b8, a3 x b9, 0, 0)
(a4 x b0, a4 x b1, a4 x b2, a4 x b3), (a4 x b4, a4 x b5, a4 x b6, a4 x b7), (a4 x b8, a4 x b9, 0, 0)
(a5 x b0, a5 x b1, a5 x b2, a5 x b3), (a5 x b4, a5 x b5, a5 x b6, a5 x b7), (a5 x b8, a5 x b9, 0, 0)
(a6 x b0, a6 x b1, a6 x b2, a6 x b3), (a6 x b4, a6 x b5, a6 x b6, a6 x b7), (a6 x b8, a6 x b9, 0, 0)
(a7 x b0, a7 x b1, a7 x b2, a7 x b3), (a7 x b4, a7 x b5, a7 x b6, a7 x b7), (a7 x b8, a7 x b9, 0, 0)
(a8 x b0, a8 x b1, a8 x b2, a8 x b3), (a8 x b4, a8 x b5, a8 x b6, a8 x b7), (a8 x b8, a8 x b9, 0, 0)
(a9 x b0, a9 x b1, a9 x b2, a9 x b3), (a9 x b4, a9 x b5, a9 x b6, a9 x b7), (a9 x b8, a9 x b9, 0, 0)

Ⅴ. 암호화 모듈의 성능 분석

본 논문에서는 지금까지 ARM을 통해 제안된 기법들과의 성능을 비교해 봄으로써 해당 논문의 성능의 우수함을 확인해 보도록 한다. 본 논문에서는 2011년도에 ARM A8을 사용하여 구현한 논문결과를 참조하여 비교하도록 한다[18]. 이에 대한 자세한 사항은 [표 3]에 나타나 있으며 동일한 비트 크기로 구현되지 않아 연구결과를 통해 예측되는 결과값을 표시하였으며 옆에 (e)와 같이 표기되어있다. 제안된 알고리즘은 160비트 상에서 구현되어 나타났으며 256, 446비트에 대한 결과값은 예상되는 결과값을 비트의 크기의 비율에 따른 소모되는 연산의 크기로 하여 계산하였다. 성능 비교 결과 본 논문에서 제안한 곱셈 구조를 통해 연산을 수행하게 될 시 보다 높은 성능이 나타남을 확인할 수 있다. 비록 본 논문에서는 160비트에 대한 결과값만을 제공하였지만 더 큰 비트에 대한 구현도 동일한 구조를 통해 연산이 가능하다. 따라서 본 논문의 결과는 보다 큰 비트에도 적용되어 사용되는 것이 가능하다.

표 3. 제안된 기법의 성능 비교
Table 3. Performance Comparison of proposed method

Bit size	[18]	Proposed
160	794(e) (clock)	500 (clock)
256	1710 (clock)	1076(e) (clock)
446	4010 (clock)	2525(e) (clock)

Ⅵ. 결 론

본 논문에서는 안드로이드 상에서 보다 효율적인 공개키 기반 암호화 구현을 위해 NDK 상에서의 어셈블리(assembly) 코딩 기법과 최신 NEON 기능을 통해 큰 인수 상에서의 곱셈(multiplication)기를 구현하였다. 해당 구현을 통해 JAVA상에서 제공하는 패키지보다도 효율적인 곱셈(multiplication)이 가능함을 확인할 수 있었다. 이는 기계어의 최적화와 NEON 기능의 병렬 연산을 통해 보다 향상된 성능이 나타남을 확인할 수 있었다. 해당 제안 기법은 신호 처리, 그래픽 그리고 암호화와 같은 다양한 분야에 적용되어 안드로이드 상에서 보다 효율적인 연산 구현에 적용될 수 있다고 생각된다.

References

[1] Google cooperation, *"Android developer,"* 10, 29, 2012, http://developer.android.com/index.html

[2] Java cooperation, *"Java API,"* 10, 29, 2012, http://developer.android.com/reference/java/math/BigInteger.html.

[3] Comba, P., "Exponentiation cryptosystems on the IBM PC," *In: IBM Systems Journal* 29(4), pp. 526-538. 1990

[4] Google cooperation, *"Android.com,"* 10, 29, 2012, http://www.android.com

[5] Google cooperation, *"Android SDK | Android Developers,"* 10, 29, 2012, http://developer.android.com/sdk/index.html

[6] Google cooperation, *"Dalvik virtual machine insights,"* 10, 29, 2012, http://www.dalvikvm.com/

[7] Google cooperation, *"Android NDK | Android Developers,"* 10, 29, 2012, http://developer.android.com/sdk/ndk/index.html

[8] Wikipedia, *"Java Native Interface - Wikipedia,"* 10, 29, 2012, http://en.wikipedia.org/wiki/Java_Native_Interface

[9] Rob Gordon, *"Essential JNI: Java Native Interface,"* ISBN 978-0136798958, Jun. 1998.

[10] Mark Allen Weiss, *"C++ for Java Programmers,"* ISBN 978-0139194245, Oct. 2003.

[11] Leonid Batyuk, Aubrey-Derrick Schmidt,Hans-Gunther Schmidt, Ahmet Camtepe and Sahin Albayrak, "Developing and benchmarking native linux applications on android, *Proceedings of the 2nd international conference on mobile wireless middleware, Operating systems, and applications (Mobilware 2009),* pp. 381-390, Beriln, Germany, pp. 28-29, Apr, 2009.

[12] Sangchul Lee and Jae Wook Jeon, "Evaluating performance of android platform using native C for embedded systems," *International Conference on Control, Automation and Systems,* Gyeonggi-do, Korea, pp. 27-30, Oct, 2010.

[13] Jung Ha Paik, Seog Chung SEO, Yungyu Kim, HwanJin Lee, HyunChul Jung, DongHoon Lee, "An efficient implementation of block cipher in android platform," *2011 Fifth FTRA International Conference on Multimedia and Ubiquitous Engineering*. pp. 173-176, June. 2011.

[14] Google cooperation, *"What is android."* 10, 29, 2012, http://developer.android.com/guide/basics/what-is-android.html

[15] Google cooperation, *"What is ndk."* 10, 29, 2012, http://developer.android.com/sdk/ndk/overview.html

[16] Nils Gura, Arun Patel, Arvinderpal Wander, Hans Eberle, Sheueling Chang Shantz, "Comparing elliptic curve cryptography and RSA on 8-bit CPUs," *6th International Workshop on Cryptographic Hardware and Embedded Systems,* pp. 119-132, Jun, 2004.

[17] Michael Hutter and Erich Wenger. "Fast multi-precision multiplication for publickey cryptography on embedded microprocessors," *Cryptographic Hardware and Embedded Systems-CHES 2011*, Volume 6917 of Lecture Notes in Computer Science, pp. 459-474. Jun. 2011.

[18] Tolga Acar Kristin Lauter, Michael Naehrig, and Daniel Shumow, *"Affine pairing on ARM,"http://eprint.iacr.org/2011/039.pdf,* Jun. 2011.

[19] Hwajeong Seo, Howon Kim, "Research on Symmetric Cryptography based on NDK," *The Korea Institute of Communications and Information Sciences, Winter Conference,* Feb, 2012.

서 화 정 (Hwa-jeong Seo)

2010년 2월 부산대학교 정보컴퓨터공학과(공학사)
2010년 2월~2012년 2월 부산대학교 컴퓨터공학부 석사
2012년 3월~현재 부산대학교 컴퓨터공학부 박사
<관심분야> 정보보안, RFID/USN, 암호 이론, VLSI 설계

김 호 원 (Ho-won Kim)

1993년 2월 경북대학교 전자공학과(공학사)
1995년 2월 포항공과대학교 전자전기공학과(공학석사)
1999년 2월 포항공과대학교 전자전기공학과 (공학박사)
2008년 2월 한국전자통신연구원(ETRI) 정보보호연구단 선임연구원 / 팀장
2008년 3월~현재 부산대학교 정보컴퓨터공학부 교수
<관심분야> 스마트그리드 보안, RFID/USN 정보보호 기술, PKC 암호, VLSI 설계, embedded system 보안