ARM to x86 바이너리 변환 기술[1]

최민*, 이원재[2] 배성준[2] 이현우[2]

*충북대학교 정보통신공학과
[2]한국전자통신연구원,
e-mail : miin.chae@gmail.com

ARM to x86 Binary Translation Techniques

Min Choi*, Wonjae Lee[2], Sungjoon Bae[2], Hyunwoo Lee[2]
*Dept. of Information and Communication Engineering, Chungbuk National University
[2]Electronics and Telecommunication Research Institute

요 약

최근 각종 스마트 디바이스의 활용이 급속히 활용이 늘어나고 있다. 본 연구는 ARM to x86 바이너리 변환(Binary Translation) 기술을 통해 인텔의 x86 기반 ATOM 모바일 프로세서에서 ARM target 으로 컴파일된 NDK 활용 안드로이드 애플리케이션을 실행하는 것을 목표로 한다. 본 논문에서는 ARM to x86 관련 바이너리 변환 기존 연구를 분석한 후, 실제적인 ARM to x86 바이너리 변환 플랫폼을 통해 바이너리 변환 사례를 소개한다.

1. 서론

기존의 클라이언트-서버 분산 구조에서는 웹 서비스를 통해 서비스 기반 구조(Service Oriented Architecture)를 구축하고 개발생산성을 향상하는 목적으로 웹서비스를 활용한다. 스마트폰 애플리케이션의 경우는 자체적인 컴퓨팅 파워가 떨어지기 때문에, 계산시간이 많이 걸리는 작업 혹은 방대한 데이터베이스의 자료를 활용하는 작업의 경우 클라우드 컴퓨팅 서비스를 활용할 필요가 있다. 이렇게 하면 스마트폰에서는 사용자 인터페이스 중심의 코드만 실행하는 반면, 복잡한 계산 또는 계산 시간이 오래 걸리는 작업은 고성능의 클라우드 컴퓨팅 환경을 통하여 처리하고 그 결과값만 받아오는 형태로 구현할 수 있다.

또한, 이러한 클라우드 컴퓨팅 플랫폼은 현재 스마트폰 운영체제의 파편화(fragmentation) 현상을 해결한다. 특히, 아이폰(iOS)과 안드로이드(android)로 양분된 시장에서 각 스마트폰 애플리케이션이 호환되지 않고 있으며, 안드로이드 내부에서는 각 버전별로 호환 문제가 발생하고 있다. 클라우드 컴퓨팅은 특정 기업 중심의 폐쇄적 모바일 OS 플랫폼을 웹 기반 오픈서비스 플랫폼으로 개방할 수 있으며 플랫폼 종속적인 어플리케이션들을 별도로 개발해야 하는 개발비용 중복을 해결할 수 있다. 본

개발 기술은 다수의 사용자에게 저가형 모바일 단말을 대여형식으로 제공하는 다양한 서비스 모델(예: 클라우드 기반 저비용 스마트 모바일 교육 서비스, 관광 특구를 중심으로 한 스마트 투어 서비스 단말 등)에 적용하여 문화, 관광, 스마트워크(smart work), N-스크린 서비스 활성화에 기여할 수 있다.

현대 스마트폰에서 안드로이드폰, 아이폰, 윈도우폰 애플리케이션은 ARM 명령어 세트(instruction set)에 기반한 프로세서에서 동작하는 경우가 대부분이다. ARM 아키텍쳐는 물론 저전력성, 고성능, 고집적 패키징 등 여러가지 면에서 장점을 갖고 있어 현대 임베디드 시스템, SoC 분야에서 매우 널리 활용되고 있다. 현재 ARM 기반 프로세서가 대부분의 스마트폰 시장을 점유하고 있는데 반하여, 향후 x86 기반 프로세서가 탑재된 저렴한 스마트폰이 출시될 수 있는 가능성을 제시한다. 이러한 환경에서는 안드로이드 운영체제가 ARM 계열이 아닌 x86 계열 모바일 프로세서에서 운용하는 경우를 가정해야 하는데, 이 때, 애플리케이션 호환성 및 성능 문제가 발생하므로 이를 해결하기 위한 바이너리 변환 기법이 제공되어야 한다. 일반적인 안드로이드 애플리케이션은 대부분의 자바 바이트코드(JAVA bytecode)로 compile 되어 있으며 JAVA 기반의 Dalvik-VM 에서 동작하므로, x86 기반 안드로이드 운영체제에서도 문제없이 수행 가능하다, 그러나, 현재 앱스토어에 등록되어 있는 일부 애플리케이션들은 JAVA Bytecode 외에 추가적으로 NDK를 통하여 컴파일된 코드를 포함하고 있는데, 이 부분이 ARM architecture 를 target 으로 한 binary 인

[1] This work was jointly supported by the KCC(Korea Communications Commission), Korea, under the R&D program supervised by the KCA(Korea Communications Agency)" (KCA-2012-12912-03003)

경우 x86 기반 안드로이드 운영체제에서 수행하는데 어려움이 있다.

현재의 안드로이드 NDK 는 ARM 과 x86 아키텍처 모두 지원하는 방향으로 결정되었고, 이미 두 가지 (ARM target 과 x86 target)버전의 NDK 가 제공하고 있다. 따라서, 과거에 NDK 를 활용하여 개발된 일부 애플리케이션들에 대해서만 binary translation 등의 방법으로 변환 작업을 하면 된다. 따라서, 추후에 개발되는 애플리케이션 들은 NDK 를 사용하더라도 두 버전을 모두 지원하는 방향으로 정리될 가능성이 있다.

본 논문의 나머지 부분은 다음과 같다. 2 장은 본 연구와 관련된 내용을 설명한다. 3 장은 REST 오픈 API 성능 분석의 세부사항에 초점을 두고 있다. 4 장에서는 결론 및 향후 연구 방향에 관해서 소개한다.

2. 국내외 바이너리 변환 기술동향

과거 ARM architecture를 target으로 컴파일(compile)된 바이너리(binary)를 실행할 수 있도록 하는 에뮬레이션(emulation) 환경을 개발하려는 노력이 있다. 동적 변환(dynamic translation, emulation)과 정적 변환(static translation) 방법이 그것인데, 동적 변환 방법은 실행시간에 on-the-fly로 ARM 바이너리(binary)를 명령어를 x86 명령어 세트(instruction set)로translation 해준다. 이러한 과정에서 시스템 호출(system call) 역시 x86 호스트(host)의 시스템 호출(system call)로 변환된다.

- 동적 바이너리 변환 (dynamic BT)

Emulation(dynamic translation) 은 사실상 하드웨어를 소프트웨어적으로 구현하여 실제의 동작과 동일하거나 유사한 결과를 얻도록 하는 방법이다. Emulator는 CPU, system chip(DMA, timer, interrupt controller, bus controller 등) 다양한 hardware component, subsystem를 포함한다. 이러한 소프트웨어 component 의 조합은 종종 virtual machine이라 알려진다.

Virtual machine 은 가장 대표적인 방법이고 현재 널리 사용되고 있다. 호스트 플랫폼(Host platform, hardware machine)에 가상화 소프트웨어(virtualization software)를 추가함으로써, 가상머신상에서 응용프로그램(application)을 실행할 수 있도록 하는 것이다. 이러한 가상머신은 Process Virtual Machine과 (whole) System Virtual Machine 두 가지가 있다.

(1) 프로세스 VM

게스트 프로세스(guest process)가 호스트 프로세스(host process)와 서로 혼재하는 구조로서, HW 플랫폼으로부터 다른 ISA(instruction set architecture, 즉 HW 플랫폼과 다른 아키텍쳐를 target으로 컴파일된)를 갖는 애플리케이션을 실행할 수 있다.

위 좌측 그림에서와 같이 런타임 시스템(Runtime system) 수준에서 ABI level에서 coupling한다. 이러한 방법은 3D 가속을 위한 API 수준에서 구현하는데 유용하며, 적은 메모리를 소모하고 성능상 유리하다 (메모리 접근 시 MMU 변환을 수행할 필요가 없으므로). Host OS의 기능을 활용할 수 있어 유리하다.

(2) 시스템 VM (whole System VM)

시스템 VM은 전체 system environment를 제공하고, ISA 수준에서(즉, 애플리케이션은 물론 OS를 포함한 전체 소프트웨어) 가상 머신을 생성할 수 있다. 가장 대표적인 것으로 VMware, VirtualBox 등이 이에 해당한다. 구조적으로 단순하여 구현에 용이하다. 상용 또는 공개소스 소프트웨어로 널리 사용되고 있으며, 대규모 클러스터 또는 클라우드 컴퓨팅 환경에서 서버 자원활용률을 증가시키는 콘솔리데이션(consolidation)을 실현할 수 있다.

그 외에도 고수준언어 가상머신(High-Level Language Virtual Machine)이 있다. 가장 대표적인 것이 최근 안드로이드 운영체제에서 활발하게 활용되고 있는 Dalvik Virtual Machine, MS CLI(Common Language Infrastructure)가 있다. 고수준언어 가상머신은 다음 그림과 같이 운영체제와의 상호작용이 가상 플랫폼의 일부인 API를 통해서 추상화 되는 형태이다.

달빅(Dalvik) 가상 머신은 레지스터 머신 형태의 (register-based) 가상 머신이다. 댄 본스타인이 다른 구글 엔지니어들의 도움 하에 설계/구현하였다. 달빅 가상 머신은 적은 메모리 요구 사양에 최적화되어 있다. 밑에 깔린 프로세스 아이솔레이션(process isolation), 메모리 관리, 스레딩(threading) 지원 등 운영체제의 지원에 의존하나, 여러 개의 달빅 VM 인스턴스가 동시에 존재할 수 있다. 달빅 가상 머신은 종종 자바 가상 머신으로 불리우고 있으나, 엄밀하게는 서로 다르다.

달빅 가상 머신이 인지하는 바이트코드는 자바 바이트코드가 아니기 때문이다. 안드로이드 SDK에 함께 들어 있는 dx 툴(tool)을 이용하면 자바 클래스 파일들을 .dex 포맷(일종의 다른 클래스 포맷)으로 바꿀 수 있다. 다음 그림은 실제적인 Dalvik 바이트코드 포맷(bytecode format)이다.

- 정적변환 (Static BT)

정적변환(static translation)은 binary를 읽어서 원래와 동일하거나 원래의 실행과 정확하게 일치하는 결과를 얻고자 하는 목적으로 새로운 target architecture에 적합한 코드를 작성하는 기법이다. 이러한 binary translation은 binary 코드를 parsing하고 bit pattern을 target machine에서 실행 가능하도록 해야 한다. 이러한 작업은 주로 프로그램을 실행하지 않는 상태(offline)에서 이뤄지므로, 정적 변환(static translation)이라 불린다. 정적 변환(static translation) 은 몇 가지 문제점이 있다. 정적 변환기(static translator)는 자기 변경적(self-modifying) or 자기 참조적(self-referential) 코드(code)의 경우 변환(translation) 할 수 없다. 실제

로, 일부 dynamic translation (JIT 등)은 실제로 자기 변경적(self-modifying) 코드(code)를 사용한다. 즉, 실행하는 동안에 자신의 실행코드를 다시 생성한다.

다음은 ARM to x86을 위한 바이너리 변환 관련 국내외 기술개발현황을 조사한 것이다. 현재 대부분의 스마트폰 ARM 프로세서 아키텍쳐를 탑재하고 있다. TI OMAP, Qualcom, 삼성 등 대부분의 스마트폰 Application Processor(AP) 제조사들이 ARM SoC 기반으로 CPU를 제작하고 있기 때문이다. 이에, Intel과 MIPS는 ARM이 선점하고 있는 안드로이드 시장을 탈환하기 위해서 최근 모바일용 프로세서를 출시하고 안드로이드 스마트폰 시장에 진입하였다. 이로써, 현재 시장에 출시된 안드로이드 단말기는 ARM, Intel, MIPS의 3종이 되었다.

- Intel

인텔은 기존 데스크톱 시장에서 변함없는 왕좌에 올라 있다. 그러나, 모바일 분야에서는 ARM 계열 프로세서에 시장의 상당부분을 내어준 상태이다. 심지어, Intel 은 기존에 ARM 라이선스를 통해 제작하던 Xscale 사업부문을 Marvell에 매각하고 자체적인 모바일 프로세서 아키텍쳐를 추구하고 있다. 이에, 인텔은 최근 Medfield 아키텍쳐에 기반한 ATOM 프로세서를 내세우며 ARM 이 장악하고 있는 모바일 프로세서 부분에 의욕적으로 진입하고 있다.

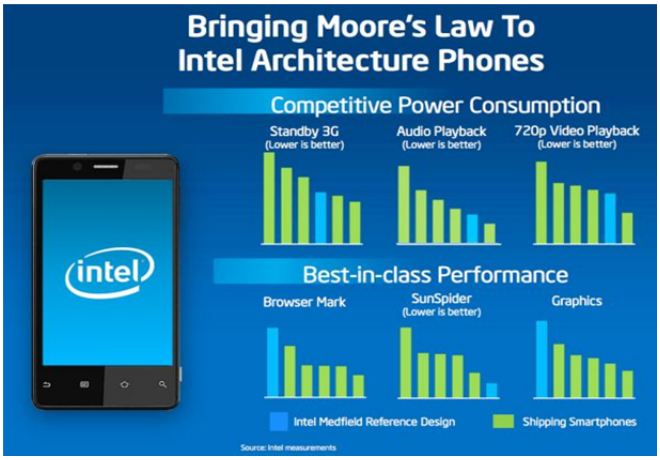

그림 1 Intel Medfield Architecture 성능분석 결과

Intel Atom 은 Ultra Low Voltage의 IA-32, x86-64 CPU로서, 넷북, 임베디드 응용, 모바일 인터넷 디바이스(MID) 등을 target으로 하는 모바일 프로세서 아키텍쳐이다.

이와 같이, Intel은 ARM architecture를 지원하는 x86 아키텍쳐를 제시하는 방향으로 안드로이드 운영체제에 대한 target architecture의 호환성 문제를 정리할 가능성이 크다.

Apple이 PowerPC architecture로부터 x86 호환 architecture로 넘어왔듯, Intel 역시 ATOM architecture를 ARM과 호환 가능하도록 하는 방향을 추진한다. Intel은 Medfield architecture를 통해 ARM binary를 직접 실행할 수 있는 기능을 제공한다. Medfield 스마트폰은 안드로이드 운영체제를 기반으로 동작하며, 안드로이드 애플리케이션은 대부분 플랫폼 독립적인 Java 바이트 코드로 되어 있으므로 ARM과 x86프로세서에서 모두 실행 가능하다. 그러나, NDK를 활용하는 일부 애플리케이션은 native ARM 코드를 사용하기 때문에 ARM 플랫폼에서만 실행된다. 이에 따라, 인텔은 바이너리 변환기(binary translator)를 통하여 x86 기반 폰에서 대부분의 안드로이드 애플리케이션이 실행 가능하도록 하는 방안을 제시하였다. 실제로 LG전자는 인텔의 Medfield 아키텍처에 기반한 최초의 인텔기반 안드로이드 폰을 공개한 바 있다.

실제로 인텔은 ATOM 프로세서 기반으로 안드로이드 운영체제를 탑재한 최초의 스마트폰으로서 XOLO라는 제품명으로 출시한 바 있다. 상기 XOLO 스마트폰에는 인텔의 ATOM Z2460 프로세서가 탑재되었는 바, 이 모델은 하이퍼쓰레딩(Hyper-Threading) 기법, 그래픽 미디어 가속(Graphics Media Accelerator) 기능, 스마트 유휴상태 전력관리 기술(Smart Idle Technology), 트레이스 기반 JIT(A trace-based JIT) 기능이 적용되어 있다.

여기서, 스마트 유휴상태 전력관리 기술이란 유휴(idle) 상태(C6 mode)에서 CPU의 전력소모 수준을 사실상 거의 0(zero) 수준까지 낮추는 기법이다. 다음은 전력소모(power)에 따른 성능(performance) 그래프를 나타낸 것이다. 이 그래프에서 전력 소모(각 CPU 모드)에 따른 성능 향상 정도를 로그스케일(log scale)로 나타내어 지므로, 비교적 낮은 전력으로 높은 성능을 얻을 수 있음을 확인할 수 있다.

Intel ATOM CPU의 또 다른 성능향상 기법은 Trace 기반의 JIT 컴파일 기법을 도입한 것이다.

원래 JIT 컴파일러는 다음과 같은 기능을 한다. 어떤 한 플랫폼에서 자바 가상머신은 컴파일된 바이트코드를 특정 프로세서가 인식할 수 있는 명령어로 해석한다. 그러나, 가상머신은 인터프리팅 방식으로 한번에 한 개의 바이트코드 명령어씩 번역한다. 그런데, 가상머신의 특정한 버전은 JIT 컴파일러 (실제로 2번째 compiler)를 사용하여, 바이트코드를 특정 시스템의 코드로(마치 그 프로그램이 처음부터 그 플랫폼에서 컴파일된 것처럼) 미리 컴파일 할 수 있다. 이와 같이 코드가 일단 JIT 컴파일러에 의해 미리 컴파일되면, 명령어를 매번 번역하며 실행하는 것 보다 대체로 더 빠른 속도로 실행할 수 있다. JIT 컴파일러는 바이트코드를 특정 플랫폼에서 즉시 실행 가능한 코드로 미리 컴파일 한다. 특히 실행 가능한 메쏘드가 반복적으로 재사용될 경우에, JIT 컴파일러를 선택하는 것이 대체로 좋은 효과를 얻는다.

이러한 JIT(Just in time) 컴파일 기법은 크게 다음과 같이 두가지로 분류된다. 하나는 트레이스(Trace) 기반 JIT이고 다른 하나는 메소드 (Method) 기반 JIT이다. 다음 그림은 Method-granularity JIT와 Trace-granularity JIT의 원리를 그림으로 표현한 것이다. 그림 왼편에는 Method-granularity JIT를 나타내고 있다.

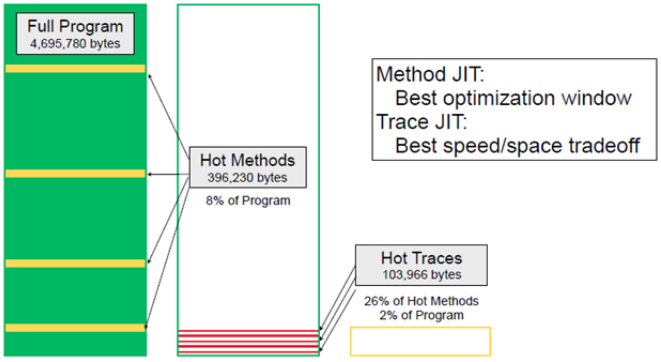

그림 2 Method-granularity JIT와 Trace-granularity Trace 개념 비교

전체 프로그램 중 일부 자주 사용되는 함수(hot method)들을 미리 컴파일 해 두는 방식이 Method-granularity JIT이다. 따라서, 자주 사용되는 함수들을 매번 인터프리팅 방식으로 번역할 필요가 없으므로 성능이 향상된다. 이 방식은 함수 전체를 컴파일하고 메모리 상에 캐쉬하여야 하므로 많은 메모리 소요량을 필요로 한다. 반면, Trace-granularity JIT는 자주 실행되는 실행경로만을 JIT 컴파일하고 이를 메모리 상에 보관한다. 이 방법은 자주 발생하는 실행패스(hot execution path)에 대해서만 명령어의 컴파일 범위를 한정하므로, 보다 적은 메모리를 소요하며 빠른 실행속도를 얻을 수 있다. 결과적으로, Trace-granularity JIT는 Method-granularity JIT에 비하여 메모리에 상주하는 데이터량이 매우 적은 장점을 가진다. 메소드 기반 JIT는 자주 실행하지 않는 명령어라 할지라도 그 메소드가 빈번하게 호출되기만 하면 모두 컴파일되어 메모리에 상주하므로 메모리 소요량이 많은 단점이 있다.

3. 바이너리 변환 구현

본 실험에서는 QEMU 1.2.0 의 소스코드를 Ubuntu 리눅스 15 에서 소스코드 컴파일하여 설치하였고, 각종 ARM 실행파일(binary)를 활용하여 x86 리눅스 운영체제에서 정상적으로 실행되는 것을 확인하였다. QEMU 1.2.0 는 다양한 아키텍쳐(ARM, x86, ppc, ppc64 등)의 바이너리(실행파일)를 실행가능하지만 동적으로 공유되는 라이브러리를 함께 보유하고 있어야만 정상적으로 실행가능하다. 이는 컴파일러가 라이브러리에서 호출되는 함수(예를들면 prinft 와 같은)를 모두 실행파일에 통합하지 않기 때문이다. 그 대신, 동적 링크 가능한 라이브러리를 통해 실행중에 필요한 라이브러리를 찾아내어 동적으로 링크하기 때문이다. 따라서, 바이너리 변환기능이 제공된다고 해서 단순하게 실행파일만 단독으로 동적 바이너리 변환함으로써 실행 가능한 것은 아니다. 실제로, ARM 바이너리를 실행하고자 한다면, ARM 기반 안드로이드 플랫폼에서 제공하는 각종 라이브러리(공유 라이브러리 파일들)가 모두 준비되어 있어야만 정상적으로 ARM 바이너리를 x86 에서 실행할 수 있다.

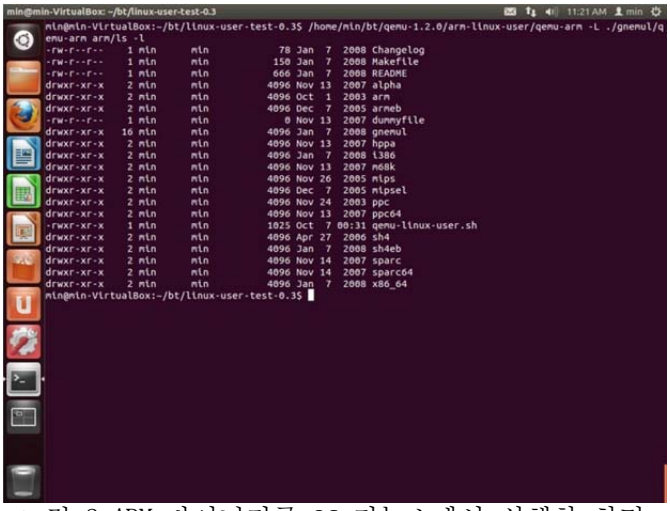

그림 3 ARM 바이너리를x86 리눅스에서 실행한 화면

그림 3 은 ARM 프로세서용으로 컴파일된 실행파일(ls)이 바이너리 변환(Binary Translation)을 통해서 x86 기반 리눅스 머신에서 실행된 화면이다. ls 명령은 ARM 프로세서에서도 현재 디렉토리내 파일들에 대한 정보를 읽어들이고 화면에 표시하는 기능을 갖고 있으므로, x86 에서 바이너리 변환하여 실행한 결과 역시 현재 디렉토리 내에서 읽어들인 파일의 정보를 화면에 표시하는 것으로 동일하다.

4. 결론

앱스토어에 등록되어 있는 기존의 안드로이드 애플리케이션들을 x86 기반 안드로이드 운영체제에서 실행하는 데 있어 제약사항이 감소한다. 이는 X86 기반 안드로이드 운영체제의 활용/보급 확대를 기대할 수 있다. 최근 인텔은 모바일 프로세서 시장에서 ARM 프로세서로부터 시장을 되찾기 위하여 ATOM 프로세서와 최초의 x86 기반 스마트폰을 출시하며 진입을 시도하고 있다. 이러한 환경에서 ARM to x86 바이너리 변환에 대한 연구는 매우 중요한 기술 중 하나이다.

참고문헌

[1] A. Chernoff, M. Herdeg, R. Hookway, C. Reeve, N. Rubin, T. Tye, S. Yadavalli, J. Yates, , FX! 32, A Profile-Directed Binary Translator, IEEE Micro

[2] K. Adams, O. Agesen, A Profile-Directed Binary TranslatorA Comparison of Software and Hardware Techniques for x86, ACM ASPLOS 2006.

[3] QEMU, http://www.qemu.org

[4] J. Martinez J. Renau, M. Huang, M. Prvulovic, J. Torrellas, Cherry: Checkpointed Early Resource Recycling in Out-of-order Microprocessors, 35th Annual International Symposium on Microarchitecture (MICRO-35),2002