모바일 폰 어플리케이션 개발환경의 구축: CPU레벨 에뮬레이터의 개발

이제형, 문수묵
서울대학교
{lordljh,smoon}@altair.snu.ac.kr

Implementation of Mobile-phone Application Develop Environment: CPU-level Emulator

요 약

현재 모바일 폰 소프트웨어 개발 플랫폼들의 에뮬레이터들은 API 레벨로만 에뮬레이션을 하는 한계로 인해 실제 폰과 완전히 동일한 동작을 보장할 수 없으므로 개발중인 어플리케이션의 동작을 제한된 수준 안에서 확인해 볼 수밖에 없었다. 하지만 CPU 레벨의 에뮬레이터가 제공된다면 실제 폰과 동일한 동작을 PC에서도 보장할 수 있게 되어 어플리케이션의 개발에 있어서 보다 수월한 테스트, 동작의 동작 특성 파악 및 성능 평가 등이 가능해 진다고 할 수 있다. 이 논문은 ARM 기반의 CPU 레벨 에뮬레이터를 제공함으로써 이와 같은 개발상의 이점을 취함과 동시에 널리 사용되는 디버거(GDB)와 연동되도록 하여 친숙하고 쾌적한 디버깅 환경을 제공할 수 있음을 보여주며, 이러한 에뮬레이터 플랫폼 개발에서 발생하는 여러가지 문제점들의 해결방안을 제시하고자 한다.

1. 서론

이동통신의 대중화로 인해 모바일 폰 어플리케이션의 개발 역시 활발히 이루어지고 있다. 하지만 폰 어플리케이션의 개발은 일반적인 PC용 소프트웨어 개발환경에 비해 열악한 환경과 여러가지 제약을 갖고 있다. 폰 어플리케이션 개발은 주로 PC환경에서 모바일 폰의 행동을 에뮬레이션 할 수 있는 플랫폼에서 이루어 지고 있다.

개발하고 있는 프로그램을 직접 모바일 폰 위에서 동작시키는 것은 거쳐야 할 절차가 복잡하고 시간이 많이 걸리기 때문에 프로그램을 다시 빌드하고 폰으로 업로드하고 프로그램의 행동양태를 지켜보는 것 자체가 개발 단계에서는 매우 번거로운 작업일 뿐 아니라 잘못된 행동을 발견했다 하더라도 디버깅 수단이 제공되지 않거나 미약하기 때문에 실제로 폰에 프로그램을 업로드 하는 것은 프로그램 개발의 가장 마지막 단계에서 이루어진다. 대신 개발 단계에서는 모바일 폰의 행동을 그대로 흉내 낼 수 있는 에뮬레이터를 PC에서 동작시켜 진행하게 된다.

현재 플랫폼들의 에뮬레이터들은 API 레벨 에뮬레이션 정도만을 지원하고 있다. 즉 API 이하 레벨, 말하자면 운영체제(OS)나 가상머신(VM)은 실제로는 PC용으로 컴파일 되고 PC에서 동작하는 프로그램이기 때문에 실제 폰과 완전히 동일한 동작을 보장할 수는 없다. 따라서 개발중인 어플리케이션의 동작을 제한된 수준 안에서 어느 정도의 오차를 감안하며 확인해 볼 수밖에 없다. 하지만 CPU 레벨의 에뮬레이터, 즉 OS나 VM을 위시한 모든 환경이 에뮬레이션 될 수 있다면 실제 폰과 동일한 동작을 PC에서도 보장할 수 있게 되어 어플리케이션의 개발에 있어서 보다 수월한 테스트, 동작의 특성 파악, 성능 평가 등이 가능해 지게 된다.

이 논문은 ARM 기반의 CPU 레벨 에뮬레이터를 개발함으로써 이와 같은 개발상의 이점을 취함과 동시에, 널리 사용되는 디버거(GDB)와 연동이 가능하도록 하여 친숙하고 쾌적한 환경을 제공할 수 있음을 보여주며, 이러한 에뮬레이터 개발에서 발생하는 여러 문제점들에 대한 해결방안을 제시하고자 한다.

2. 구현목표

개발하고자 하는 CPU 레벨의 ARM 에뮬레이터는 ARM의 상용 컴파일러인 ADS에서 생성된 바이너리를 완벽하게 수행할 수 있는 프로그램이 되어야 한다. 우리는 ARM 에뮬레이터 소스 가운데에서 어느 정도 완성도를 가지고 있으며 GDB와 연동이 가장 쉽게 이루어질 수 있는 GDB 내장 에뮬레이터를 수정하여 에뮬레이터를 작성하였다. GDB 내장 ARM 에뮬레이터의 경우 THUMB 모드에 대한 구현도 잘 되어 있는 장점이 있다.

ARM 기반의 CPU 레벨 에뮬레이터 개발을 위해서 다

이 논문은 2006년 정부(교육인적자원부)의 재원으로 한국학술진흥재단의 지원을 일부 받아 수행된 연구임 (KRF-2006-311-D00757).

음의 주요 두 가지 항목에 대한 연구가 제안되고 수행되었다.

2.1 HAL(Hardware Abstraction Layer)과의 연동

에뮬레이터가 포함된 플랫폼은 폰 어플리케이션을 수행할 수 있는 추가적인 장치가 마련되어 있어야 하는데, 폰의 low level 서비스를 제공하는 HAL 라이브러리와 연동될 수 있어야 한다. HAL 라이브러리는 모바일 폰의 하드웨어적인 동작을 PC상에서 시뮬레이션하기 위한 것으로 윈도우즈(Windows)용으로 개발된 C 라이브러리이며 단말기 개발업체가 제공한다. 에뮬레이터는 작성된 어플리케이션에게 이 라이브러리의 코드를 서비스하여 작동이 가능하게 한다.

2.2 GDB와의 연동

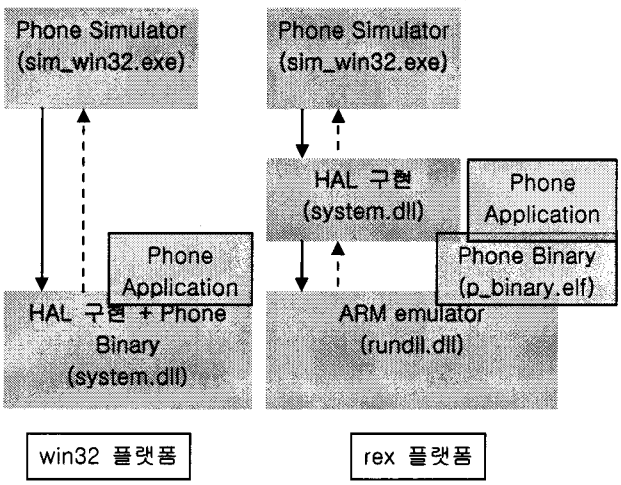
GDB를 이용해 폰에서 동작하는 바이너리의 디버깅을 가능케 한다. 우리의 구현은 내부적으로 GDB와 에뮬레이터가 TCP serial 통신을 통해 연결되어 원격 디버깅 형태로 동작하지만 GDB와 에뮬레이터 모두 같은 컴퓨터에서 동작하므로 일반적으로 GDB 를 사용할 때와 크게 다르지 않다.

3. 플랫폼의 구성

본 연구는 윈도우즈에서 유닉스(UNIX) 환경을 제공하는 cygwin[2]상에서 동작하는 에뮬레이터 플랫폼을 가정하고 진행되었다. 이미 API 레벨에서 동작하는 에뮬레이터를 보유하고 있는 상황에서 새로이 CPU 레벨에서 동작하는 에뮬레이터를 개발하고자 하였다. 기존의 에뮬레이터 플랫폼을 win32 플랫폼, 새로운 CPU 레벨 플랫폼을 rex 플랫폼이라 부르기로 한다. (rex는 모바일 폰에서 동작하는 OS의 명칭이다.) 또한 우리는 폰에서 자바(Java)프로그램을 동작시키기 위한 자바 VM 환경을 염두에 두고 있다.

<그림 1>에서 보는 바와 같이 본래의 win32 플랫폼은 에뮬레이터의 GUI 인터페이스 및 폰 바이너리의 기동을 담당하는 폰 시뮬레이터(sim_win32.exe), 그리고 이 폰 시뮬레이터에 의해 실행되는 dll 형태의 폰 바이너리(system.dll)로 구성된다. HAL 구현은 system.dll에 함께 포함되어 있다. 여기서 폰 바이너리라 함은 자바 VM을 의미하며, API 레벨의 에뮬레이터이기 때문에 폰 바이너리 역시 윈도우즈 바이너리라는 점을 주목해야 한다. 폰 어플리케이션은 자바 바이트코드이기 때문에 플랫폼과는 무관하다.

새로이 구축한 rex 플랫폼은 비슷한 방법으로 폰 시뮬레이터가 system.dll을 호출하도록 하되 system.dll은 오직 HAL 구현만을 가지고 있으며 나머지 폰 바이너리에 해당하는 코드는 ARM 바이너리(p_binary.elf)로 빌드된다. 이렇게 ARM 바이너리로 빌드된 폰 바이너리는 에뮬레이터에 의해 로딩되고 실행되는데 에뮬레이터는 dll 형태로(rundll.dll) system.dll에 의해 호출된다. CPU 레벨 에뮬레이터이기 때문에 폰 바이너리 역시 ARM 바이너리인 점을 주목하자. 다만 HAL 라이브러리는 윈도우즈 라이브러리이기 때문에 폰 바이너리처럼 ARM 바이너리로 제공되지 않는다.



<그림 1> 양 플랫폼의 구성. 화살표는 서비스 호출을 의미하며 점선은 callback호출을 의미한다.

ARM 에뮬레이터는 때때로 폰 바이너리가 요청한 HAL 서비스를 제공받기 위해 system.dll 측의 함수를 호출해야 하는데, 이것은 미리 system.dll 에서 HAL 함수들의 포인터들을 ARM 에뮬레이터로 넘겨주는 과정을 통해 가능해진다. 이러한 기법을 callback 메커니즘이라 한다. <그림1>에서 점선으로 된 화살표는 이러한 callback 호출을 의미한다.

이제 두 가지 구현 목표에 대한 해결방법에 대해 설명하도록 한다.

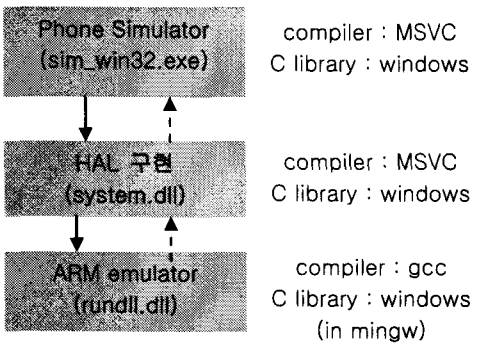
4. HAL 연동을 위한 과제 및 호환성 문제의 해결

4.1 DLL 비호환성 문제 해결

이미 유닉스 환경을 제공하는 툴로 개발자들에게 잘 알려져 있는 cygwin[2]은 대신 프로그램을 작동시키기 위해 몇 가지 제약사항이 존재한다. 가장 중요한 것은 윈도우즈 환경에서 일반적인 컴파일러 (예: MSVC)를 이용해 생성한 바이너리는 cygwin 환경에서 생성한 바이너리를 호출하는데 문제가 있다는 점이다. 이것은 시스템 라이브러리의 비호환성 때문에 발생한다.

이러한 비호환성이 문제가 되는 것은 <그림1>을 보면

이해할 수 있다. win32 플랫폼에서는 Phone Simulator 나 system.dll 모두 MSVC를 이용해 생성되어 제공되기 때문에 아무런 문제가 없다. 하지만 rex 플랫폼의 경우 ARM 에뮬레이터는 GNU 소스 가운데서 GDB 내장 에뮬레이터를 수정해 만들어 졌다. 하지만 현재로서는 GDB 소스를 MSVC에서 빌드하는 방법은 제공되지 않기 때문에[4] gcc를 이용해 cygwin 환경에서 빌드하여야만 한다. 따라서, GDB를 빌드하기 위해, 그리고 cygwin에 종속된 라이브러리 문제를 해결하기 위해 또다른 UNIX 기반의 환경을 모색하게 되었고, mingw라는 환경에서 해답을 찾을 수 있었다.



<그림2> mingw를 이용한 바이너리 호환성 해결

mingw(Minimalist GNU for Windows)는 cygwin과 비슷하지만 그보다 매우 간단하고 작은 환경이며 GNU 툴들을 이용하면서도 윈도우즈 라이브러리를 그대로 사용할 수 있는 환경이다.[3] 하지만 아직 mingw 환경에 성공적으로 포팅된 cross-gdb는 보고된 사례가 없다. 왜냐하면 mingw측에 구현되지 않은 API들이 다수 존재하기 때문인데 주로 시리얼 통신에 관련된 것들, RDI/RCP를 지원하기 위한 API, 그리고 대부분의 UNIX 기반 singal 들이 지원되지 않고 있기 때문이다. 하지만 문제가 되는 GDB 소스들 가운데에서 에뮬레이션에만 필요한 부분을 추출하여 대체하여야 할 API들을 최소화함으로써 동작하는 버젼을 완성할 수 있었다.

4.2 ADS 와 GNU 간의 바이너리 호환성 해결

ADS와 ARM용 GNU 툴은 동일한 ARM 아키텍쳐, 동일한 ELF 포맷을 따름에도 불구하고 내부적인 구현이 달라 바이너리 간의 호환성이 보장되지 못한다. ADS에서 생성한 아주 간단한 바이너리라 할지라도 ARM용으로 생성된 GDB의 에뮬레이터에서 작동되지 않는다. 이러한 비호환성 문제는 어느 한쪽이 표준을 제대로 따르지 않았거나, 다수의 선택 가능한 표준이 존재하기 때문에, 혹은 표준을 따르긴 하되 허용하는 선택사항이 다수 존재하기 때문에 발생하는 문제이다.

가장 대표적인 비호환성의 이유는 SWI(Software Interrupt) semi-hosting 서비스 루틴 때문이다.[1] ADS가 생성하는 코드에서 삽입되는 SWI 요청 코드에 대응하는 서비스 중 몇 가지가 GDB의 ARM 에뮬레이터 소스에 구현되어 있지 않은데(예: SYS_ISTTY) gcc에서 생성하는 코드는 해당 semi-hosting SWI를 생성하는 경우가 없기 때문에 전혀 문제가 되지 않지만 ADS에서 생성한 코드는 그렇지 않다. 따라서 이러한 부족한 구현을 찾아서 추가함으로써 문제점을 해결할 수 있다.

4.3 디버깅 정보 호환성 해결

ADS와 GNU 툴 공히 디버깅 정보로 DWARF2 포맷을 따르고 있다. 하지만 ELF 바이너리에 호환성 문제가 있던 것처럼 ADS에서 생성한 바이너리는 GDB에서 곧바로 이용될 수 없다.

DWARF2는 하나의 함수를 compilation unit(cu)이라고 불리는 단위로 관리하는데[5], gcc와는 달리 ADS 컴파일러는 하나의 함수에 대해 2개의 cu를 생성한다. 이때 선행하는 cu는 후행하는 cu의 완전한 부분집합이며, 실제 실행에 필요한 모든 정보를 담고 있는 cu는 후행 cu 임에도 불구하고 변수의 base type 정보는 항상 선행하는 cu에서 참조하도록 되어있다.

본래 GDB에서 cu 내부의 각 entry를 해석하기 위해서는 전체 cu를 전부 읽어서 해석해야 했으나 이와 같은 base type 정보를 가져올 경우에는 선행하는 cu를 전부 해석할 필요 없이 필요한 type 정보만을 가져올 수 있도록 수정하였다.

4.4 HAL 라이브러리 연동

에뮬레이터가 ARM 코드의 수행 중 HAL 서비스를 요청받게 되면 에뮬레이터는 이것을 적절히 감지하여 system.dll에 존재하는 서비스 함수를 호출해야 한다. 즉, ARM 코드로 된 함수가 아닌 호스트(PC) 상에 존재하는 함수를 호출하는 메커니즘이 제공되어야 한다. 그러기 위해서는 이러한 HAL 함수가 일반적인 함수 호출이 아닌 특별한 경우라는 것을 '적절히' 판단하는 방법이 필요한데 그래서 선택한 방법이 등록되지 않은 SWI를 이용한 방법이다.

ARM 코드로 생성되는 폰 바이너리에 링크시킬 목적으로 HAL 함수와 동일한 이름을 가지고 있지만 함수 본체는 단지 SWI 명령만으로 이루어진 dummy HAL 함수를 만든다. 에뮬레이터에서는 이 SWI에 대응하는 서비스 루틴을 통해 어떤 HAL 함수가 요청되었는지를 파악하여 실제 HAL 함수를 호출하게 된다. 이러한 과정은 일반적인 SWI handler의 수행과정과 동일하며 이 서비스 루틴(HAL 함수)이 수행을 끝내면 곧바로 호출했던 ARM 코드로 되돌아간다.

가장 먼저 살펴볼 것은 dummy HAL 함수들이다.

dummy HAL 함수는 다음과 같은 형태의 어셈블리 코드로 작성되어 있으며 모든 HAL 함수들의 목록을 빠짐없이 포함하고 있어야 하고 폰 바이너리를 생성할 때 함께 링크되어야 한다.

```
hal_ui_control_display PROC
        ;; hal_Int32
        ;; hal_ui_control_display(hal_Int32 dev_idx, hal_Int32 arg1, hal_Int32 arg2, void *arg3, hal_Int32 arg3_length)
        SWI 0xAC ;; 0xAC SWI 를 통해 HAL 서비스 요청이 들어왔음을 알린다.
        ;; 에뮬레이터 내에서 hal_ui_control_display 에 해당하는 서비스를 수행한다.
        MOV PC, LR      ;; 서비스를 마치고 return
        DCW 0x77        ;; 어떤 HAL 함수인지를 알려주기 위한 식별자
        ENDP
```

에뮬레이터에서는 SWI가 호출되었을 때 PC가 가리키는 주소의 메모리 값, 즉 [PC]를 얻어오면 DCW가 정의한 값을 얻을 수 있다. ARM 코드에서 PC 값은 항상 2단계 앞의 word를 가리키고 있기 때문이다.(ARM 모드에서는 +8, THUMB 모드에서는 +4)

SWI가 0xAC인 경우 HAL 서비스가 요청된 것으로 판단하고 [PC]의 값을 불러 와 어떤 HAL이 호출되었는지 새로운 switch문을 통해 판별한다. 각 case문에서 실제로 일어나는 작업은 실제 HAL 함수에 대해 적절히 argument를 가공하여 넘겨주고 그 return 값을 받아오는 일이다.

5. GDB 연동

5.1 구현 방법의 선택

mingw 환경에서 빌드한 GDB 소스는 에뮬레이터 부분만 활성화 되어 있을 뿐 디버거는 동작하지 않도록 되어 있다. 만약에 mingw 환경에서든, 혹은 cygwin 환경에서든 GDB의 에뮬레이터와 디버거가 모두 동작하도록 빌드가 가능했다고 하더라도 현재의 플랫폼 구성을 놓고 볼 때 에뮬레이터가 디버거와 연동하기 위해서는 에뮬레이터가 디버거를 호출하고 컨트롤할 수 있어야 한다. 하지만 본래 GDB의 구현은 에뮬레이터가 디버거에 종속적인 형태로 작성되어 있기 때문에 원래의 GDB 구현을 그대로 이용할 수는 없다. 따라서 디버거를 에뮬레이터와 독립적으로 동작시킬 수 있는 방법이 적합하다고 판단되어 GDB에서 제공하는 remote serial protocol을 통한 원격 디버깅을 이용하는 방법을 선택하였다.[4] ARM 용의 GDB 디버거는 cygwin 환경에서는 무리 없이 빌드가 가능하므로 이 디버거와 mingw 환경에서 빌드된 에뮬레이터가 서로 통신할 수 있도록 만들면 된다.

원격 디버깅을 구현하는 방법에도 여러 가지가 있다. 가장 먼저 생각할 수 있는 것은 ARM 표준의 RDI 인터페이스를 이용하는 것인데 앞서 언급한 대로 mingw에서는 대부분의 RDI API들이 지원되지 않는다는 근본적인 문제가 있다.

GDB 소스에 함께 배포되는 프로그램으로 GDB server란 것이 있다. 이것은 실제 디버깅 하고자 하는 target 위에서 원격으로 동작하는 작은 프로그램인데 이 프로그램을 통해 디버깅 하고자 하는 바이너리를 동작시키면 알아서 프로그램의 제어 및 원격 통신을 담당한다. 하지만 이것은 간단한 프로그램의 디버깅에는 가능하나 지금의 플랫폼은 GDB server가 제어할 수 있는 단순한 형태를 넘어섰다.

원격 디버깅의 마지막 방법으로 stub을 이용할 수 있다. 이것은 원래 디버깅 하고자 하는 바이너리에 built-in하는 모듈로서 x86, sparc용 소스가 본보기로 GDB와 함께 배포되며 기본적인 serial통신 API 및 trap handler만 구현해주면 GDB와 원격 디버깅이 가능토록 한다. 이 소스 형태의 stub은 추가해야 할 구현이 비교적 간단할 뿐 아니라 에뮬레이터에 쉽게 포함시킬 수 있어 이 stub을 ARM 용에 맞게 수정하고 윈도우즈 환경에 맞게 통신부를 구현한 다음 에뮬레이터에서 제공하는 API들을 이용한 trap handler를 구현함으로써 원격 디버깅이 가능하게 되었다.

5.2 GDB client의 수정

앞서 언급한 ADS와 GNU 툴 간의 바이너리 호환성, DWARF2 디버깅 정보 호환성에 관한 문제 이외에도 ABI, symbol table의 활용 정책 등에서 또 다른 호환성 문제가 남아있기 때문에 ADS에서 생성한 바이너리를 GDB 디버거에서 별다른 수정 없이 디버깅하기는 불가능하다. 따라서 GDB의 디버거 역시 수정된 것을 이용하여야 한다.

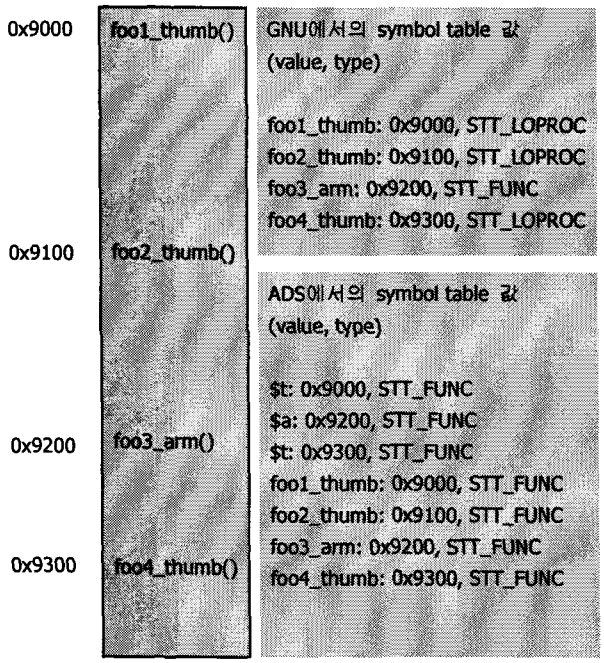
바이너리 비호환성 문제에서 제시한 문제들은 단지 에뮬레이터에서 프로그램을 구동하기 위해 필요한 조치였다고 하면 다음에 설명하는 문제들은 프로그램의 디버깅에 관련된 바이너리 상의 차이점이다.

5.2.1 ARM/THUMB 코드를 구분하는 방법

GNU의 경우 함수 이름에 해당하는 symbol의 type을 통해 ARM/THUMB 코드를 구분한다. symbol type이 STT_FUNC(2)이면 ARM 함수, STT_LOPROC(13)이면 THUMB 함수임을 의미한다.

ADS에서 모든 함수의 symbol type은 STT_FUNC(2)이다.[1] 따라서 함수 이름만으로 ARM/THUMB코드를 구분할 수 없다. 대신 부가적으로 $a, $t와 같은 mapping symbol을 사용하는데 $a는 해당 주소로부터

ARM 코드가 시작함을, $t는 해당 주소로부터 THUMB 코드가 시작함을 의미하며, 이 정보를 취합하여 어디에서 어디까지가 ARM이고 THUMB코드인지를 구분할 수 있다.



<그림3> ARM/THUMB 코드가 혼재한 예와 GNU/ADS의 symbol table 차이

<그림3>과 같이 foo3_arm() 은 ARM, 나머지 함수들은 THUMB 코드라고 할 때, GNU/ADS에서 symbol table 정책의 차이점을 볼 수 있다. GNU에서는 $a, $t 라는 mapping symbol을 사용하지 않고 직접 함수 이름의 type에 이런 정보를 기입한 반면 ADS의 경우는 mapping symbol의 정보를 통해 0x9000~0x9200까지가 THUMB, 0x9200~0x9300까지가 ARM, 그리고 0x9300 이후가 다시 THUMB 코드임을 파악한다.

GDB에서 이해가 가능하도록 ADS의 type 정보를 GNU의 그것처럼 변환하도록 수정하였는데 모든 mapping symbol을 검색해 영역별 정보를 취합한 다음 각각의 함수 이름에 해당하는 symbol의 type에 대해 THUMB 코드인 경우에는 STT_FUNC를 삭제하고 STT_LOPROC를 기입해 넣음으로써 이 문제를 해결하였다.

5.2.2 mapping/tagging symbol 의 활용

gcc에서 생성한 바이너리도 mapping symbol을 생성하긴 하지만 디버깅 정보로는 적극적으로 이용되지 않는다. 하지만 ADS가 생성한 바이너리는 ARM/THUMB을 구분하기 위한 정보로 mapping symbol $a, $t, $d를 적극적으로 이용함과 동시에 추가적으로 $b, $p, $f, $m 이라는 tagging symbol을 생성한다.[1] 이러한 mapping/tagging symbol은 GDB에서 불필요할 뿐만 아니라 그대로 둘 경우 일반적인 symbol과 뒤섞여 마치 함수 이름인 것처럼 인식되는 등 원치 않는 결과를 가져올 수 있으므로 GDB에서 이러한 symbol들을 무시하도록 하였다.

5.2.3 ABI 의 차이

함수 prologue의 instruction 구성, stack frame의 형태, register의 용도 등에서도 다수의 차이점이 있다. ATPCS(ARM/THUMB Procedure Call Standard)에서 정의한 기본적인 calling convention을 따르더라도 구현에 따른 세부적인 변경이 있을 수 있고, 또한 ATPCS 문서에도 역시 여러 가지 변종 standard 가 쓰일 수 있음을 설명하고 있다. THUMB 코드를 위주로 차이점에 대해 간략히 살펴보면 다음과 같다.

GNU의 전형적인 prologue는 다음과 같은 instruction들로 구성된다.

```
PUSH    {rlist, r7, lr}
SUB     sp, sp, #n
ADD     r7, sp, #m 혹은 MOV r7, sp
```

또는

```
PUSH    {rlist, r7, lr}
MOV     r7, sp
SUB     sp, #12
```

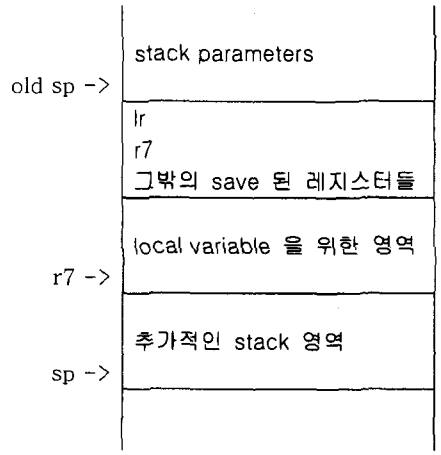
(여기서 rlist 는 "r4, r6", "r4-r7" 과 같이 표기할 수 있는 다수의 레지스터 리스트)

이와 같은 prologue에 의해 stack frame은 다음의 <그림4>와 같은 형태를 가지게 된다. GNU에서는 save된 레지스터와 local variable을 위한 영역으로 구성된 일반적인 stack frame과는 달리 추가적인 stack영역을 확보하며 이 경계는 r7을 통해 구분하고 있다. 보통은 sp와 구분하여 old sp를 저장하는 용도로 사용되는 레지스터를 frame pointer라 부르지만 여기서는 r7을 frame pointer라고 부른다.

ADS의 경우 가장 큰 차이점은 r7을 frame pointer로 사용하지 않는다는 점이다. 따라서 추가적인 stack 영역도 확보하지 않는다. 그러므로 ADS 가 생성한 전형적인 prologue 는

```
PUSH    {rlist, lr}
SUB     sp, sp, #n
```

의 단순한 형태가 된다.



<그림4> GNU에서의 전형적인 stack frame구성

하지만 ADS의 prologue를 파악함에 있어서 몇 가지 주의 사항이 있는데 GNU의 경우 prologue가 매우 정형적인 형태를 지키고 있어 문제가 없지만 ADS의 경우 최적화의 영향으로 인해 instruction의 순서가 뒤죽박죽 될 수 있다는 점, 그리고 calling convention에 의하면 r0-r3은 caller save, r4-r7은 callee save register 로 되어 있지만 r0-r3를 push하는 명령어가 사용되기도 한다는 점이다.

<표1> ADS의 변칙적인 prologue의 형태

예1	예2	예3
ldr r0,[pc,#136]	push {r4,r5,r6,r7,lr}	push {r0,r1,r2,r3}
push {r4,lr}	mov r6,r1	push {r3,lr}
ldr r0,[r0,#0]	mov r5,r0	add r0,sp,#12
sub sp, #24	mov r0, r5	str r0,[sp,#0]
	sub sp, #28	ldr r0,[sp,#8]

<표1>에서 몇 가지 변칙적인 prologue의 형태를 볼 수 있는데 ldr, mov와 같이 prologue와 상관없는 instruction들이 도중에 끼어드는 것을 볼 수 있다. 또한 예3 에서 보면 r0, r1, r2, r3이 push되고 나서 또다시 r3, lr이 push되는 것을 볼 수 있는데, 앞의 push {r0, r1, r2, r3}는 parameter로 넘어온 값들이 그 즉시 이용되지 않는 경우 미리 stack에 저장해 둠으로써 r0-r3을 확보하는 역할을 한다. push {r3, lr}의 경우 r3는 stack에 저장하는 용도가 아니라 stack frame을 확보하기 위한 역할로 사용된 것으로 결과적으로 sub sp, #n 명령을 절약하기 위한 최적화의 한 방법이다.

prologue를 구성하는 명령어의 종류나 stack frame의 형태가 중요한 이유는 breakpoint를 설정한다거나 현재 함수의 argument에 관한 정보를 얻는다거나, frame에 관련된 디버깅 명령을 수행한다거나 할 때 prologue로 부터 모든 정보를 얻기 때문이다. 예를 들어 어떤 함수에 breakpoint를 설정하는 경우 에뮬레이터가 정확히 실행을 중단하여야 하는 위치는 prologue 바로 직후의 instruction이어야 하고, 함수의 여러 variable들을 추적하는 위치 정보는 이 prologue의 정보를 토대로 검색하게 되며, stack을 unwinding하는 경우 stack frame이 어떻게 구성되어 있는지를 디버거가 정확히 알고 있어야 한다.

6. 결론

기존 API 레벨로만 에뮬레이션을 하는 모바일 폰 어플리케이션 개발 플랫폼의 경우 실제 폰과 완전히 동일한 동작을 보장할 수 없으므로 개발중인 어플리케이션의 동작을 제한된 수준 안에서 확인해 볼 수밖에 없었다. 우리는 CPU 레벨의 에뮬레이터를 제공함으로써 폰과 동일한 동작을 PC에서도 보장할 수 있게 하여 어플리케이션의 개발에 있어서 보다 수월한 테스트, 동작의 동작 특성 파악 및 성능 평가 등이 가능한 길을 열었다. ARM 기반의 CPU 레벨 에뮬레이터를 실제로 구현하여 이와 같은 개발상의 이점을 취함과 동시에 널리 사용되는 디버거(GDB)와 연동되도록 하여 친숙하고 쾌적한 디버깅 환경을 제공할 수 있음을 보여주며, 이러한 에뮬레이터 플랫폼 개발에서 발생하는 문제점들을 기술하고 해결하는 방향을 자세하게 제시하였다. 이러한 연구는 직접적으로 동일한 형태의 플랫폼을 갖고 있는 우리나라의 모바일 폰 개발환경에 직접적인 도움이 될 것이며, cross-compile을 염두에 두고 있는 이종플랫폼간의 에뮬레이션 환경 구축에도 도움이 될 것이다.

참고문헌

[1] Advanced RISC Machines Ltd. *ARM Developer Suite ver. 1.2 Technical Reference Manual*, http://www.arm.com

[2] *Cygwin*, http://www.cygwin.com

[3] *Mingw*, http://www.mingw.org

[4] *GDB*, http://www.gnu.org/software/gdb/gdb.html

[5] *DWARF2 Debugging Information Format Specification ver. 2*, http://www.arm.com/pdfs/TIS-DWARF2.pdf