CPU 독립적인 리눅스 패키지의 설계 및 구현1)

남현우○ 김수현,
한국과학기술연구원 영상미디어연구센터
namhw@imrc.kist.re.kr, suhyun.kim@imrc.kist.re.kr

Design and Implementation of CPU Independent Linux Package

Hyunwoo Nam○ Suhyun Kim
Imaging Media Research Center, Korea Institute of Science and Technology

요 약

기존 리눅스 소프트웨어 패키지는 타겟 시스템의 CPU 타입이 정해진 후 컴파일 되어 사용자에게 배포된다. 만약 설치 시스템의 CPU를 위한 패키지가 제공되지 않을 경우 사용자는 크로스 컴파일러를 이용하여 소프트웨어를 추가적으로 빌드해야 하는데 일반 사용자가 이와 같은 작업을 수행하기란 쉽지 않았다.

다른 대안으로 JVM과 같은 가상머신 기반의 소프트웨어를 작성하면 CPU에 독립적으로 패키지를 배포하고 소프트웨어를 실행할 수 있지만 네이티브 코드에 비해 성능이 떨어진다는 단점이 있었다. 본 논문에서는 가상머신의 이점을 살리면서도 네이티브 코드와 동일한 성능을 보장해줄 수 있는 OceanVM 가상머신을 사용하여 CPU 독립적인 리눅스 소프트웨어 패키지를 설계하고 구현하였다.

1. 서 론

리눅스 운영체제는 x86 CPU 기반의 PC뿐만 아니라 임베디드 계열 CPU 기반의 스마트폰, 디지털TV등과 같은 다양한 기기에서 사용되고 있다. 기존의 수많은 리눅스 패키지들을 다양한 기기에서 사용하기 위해서는 크로스 컴파일러을 이용해 CPU에 따라 패키지의 재컴파일 작업을 수행해야만 한다. 하지만 일반 사용자가 직접 패키지를 재컴파일하여 사용하기는 쉽지 않았으며, 기업 또한 포팅 작업에 많은 비용을 지불해야 했다. 최근 고성능 가상머신 기술 개발로 여러 CPU를 지원할 수 있는 실행 파일은 사용 가능해 졌지만 이를 기존의 리눅스 배포판과 패키징 포맷들과 어우러지면서 사용하기 위한 방법론이 제공되거나 증명되지 않았다.

본 논문에서는 고성능 가상머신을 이용하여 CPU 독립적으로 설치 가능한 패키지를 설계, 구현하였다. 2장에서 CPU 독립적인 패키지 시스템을 이루고 있는 관련기술들의 소개를 하며 3장에서는 구현된 시스템의 상세한 설계 구조에 대해 분석한다. 4장에서는 구현된 시스템의 성능 평가를 수행한 다음 결과를 분석하였고 5장에서는 기존 시스템들과의 장점과 실험시 도출된 단점 및 보완해야 할 사항들에 대해서 정리하였다.

2. 관련연구

하나의 소프트웨어가 다양한 CPU를 지원하기 위한 방법으로는 크로스 컴파일러를 사용하여 특정 CPU에 맞게 실행 바이너리를 생성하는 방법과 JVM(Java Virtual Machine)과 같이 가상머신을 이용하여 특정 CPU들에 의존적이지 않는 가상머신 기반의 소프트웨어를 제작하는 방법이 있다.

일반적으로 크로스 컴파일러는 상대적으로 성능이 좋은 x86 기반의 PC에서 ARM과 같이 성능이 낮은 임베디드 CPU를 타겟으로 실행 바이너리를 생성할 때 사용된다. 개인 사용자가 소프트웨어를 다른 CPU를 지원할 수 있도록 크로스 컴파일러를 이용하여 포팅 할 때 갖는 가장 큰 문제점으로는 라이브러리 참조 문제로 인하여 컴파일 에러가 발생한다는 것이다. 호스트에서는 타겟과 동일한 컴파일 환경이 보장 되도록 호스트에서는 타겟 CPU 맞게 컴파일된 라이브러리들을 모두 확보하고 있어야 한다. 그리고 해당 패키지는 타겟 CPU에 맞게 컴파일 될 수 있도록 Makefile과 같은 빌드 스크립트 파일 수정 작업이 필요하다. 하지만 개발자가 아닌 개인 사용자가 빌드 스크립트를 수정하기 위해서는 소프트웨어의 빌드 과정을 이해하기 위한 분석과정이 요구된다.

가상머신 기반의 소프트웨어의 경우 JVM을 예로 들면 Java 언어를 사용하여 소스코드를 작성한 후 컴파일 하면 JVM이 Java 명령어로 씌여진 바이트 코드를 생성한다. 자바 바이트 코드는 해당 가상머신의 명령어로 작성된 바이너리 파일이며 런타임시 JVM에 의해 네이티브 코드로 변환하여 실행된다. 따라서 JVM이 포팅만 된다면 동일한 소프트웨어를 CPU나 운영체제와 상관없이 실행시킬 수가 있다. 하지만 JVM은 Java 언어만을 지원하고 있으며 기존 C,C++코드를 직접적으로는 활용할 수 없다는 점과 런타임시 바이트코드를 JVM을 사용하여 네이티브 코드로 변환하기 때문에 기존 네이티브 코드와 비교해서 수행속도가 떨어진다는 단점이 있다.

본 논문에서는 다양한 CPU에서 동일한 리눅스 소프트웨어 패키지를 배포하기 위해 기존 방법들의 문제점들을 해결할 수 있는 새로운 접근 방법을 제시하고자 한다.

1) 본 연구는 지식경제부 및 한국산업기술평가관리원의 산업원천기술개발사업(정보통신)의 일환으로 수행하였음. [KI002119,고성능 가상머신 규격 및 기술 개발]

이를 위해 배포 패키지 파일에는 특정 CPU에 맞게 컴파일 된 실행 바이너리를 포함시키지 않는 대신 가상머신 기반의 CPU 독립적인 Bitcode파일을 포함시킨다. 실제 가상머신 바이너리가 타겟 CPU 바이너리로 변환하는 과정은 사용자가 패키지를 설치하는 시점에 이루어지게 된다. 제시하는 방법은 JVM과 달리 C, C++ 언어를 지원하며 JVM과 같이 런타임에 번역과정이 필요하지 않으므로 성능적인 측면에서 네이티브 코드와 동일한 수준으로 실행할 수 있다는 장점을 가진다. 또한 호스트와 타겟 시스템이 동일한 상태에서 설치가 이루어져 크로스 컴파일러에서 발생하던 라이브러리 참조 문제를 해결할 수 있으며, 사용자는 패키지 빌드 시스템의 이해가 없이도 기존 패키지 설치방법과 동일한 방법으로 해당 CPU에 맞게 패키지를 설치할 수 있다. 관련 연구로 리눅스 패키지 시스템과 가상머신 기술인 LLVM을 살펴보자.

2.1 리눅스용 소프트웨어 패키지

리눅스 운영체제의 경우 많은 수의 배포판을 보유하고 있으며 배포판의 종류에 따라 사용하고 있는 리눅스 커널의 버전이나 패키징 방법, 컴파일러 및 라이브러리 버전들이 조금씩 달라질 수 있다. 따라서 소프트웨어를 일반 사용자들에게 배포하기 위해서는 해당 소프트웨어에서 사용하는 라이브러리 및 환경 정보를 기반으로 설치 시스템의 의존성 검사가 요구된다. 만약 의존성 검사에서 필요한 라이브러리나 도구들이 설치되어 있지 않다면 대부분의 리눅스 배포판들은 사용자에게 설치를 요구하거나 자동으로 해당 라이브러리나 도구들의 설치를 진행한다.

구체적으로 레드햇 계열의 리눅스 배포판에서는 RPM 패키지를 사용하며 데비안 계열의 배포판에서는 DEB 패키지를 사용한다. 그 외의 배포판에서는 기존 RPM, DEB 패키지를 사용하거나 독자적인 패키지 포맷을 정의하여 사용하기도 한다. 모든 패키지 포맷들이 각자의 특징이 있겠지만 공통적으로 각 패키지 파일마다 하나의 CPU만을 지원할 수 있다. 따라서 지원하려는 CPU 개수만큼 추가적으로 패키지 제작 작업이 수행해야만 한다. 하지만 본 논문에서는 기존의 패키지 시스템을 그대로 사용하면서도 하나의 패키지 파일만으로 다양한 CPU의 지원이 가능해진다.

실험에서 사용한 리눅스 패키지 포맷으로는 데비안 계열 리눅스 배포판인 우분투 9.10 버전에서 사용된 DEB 패키지 포맷을 사용하였다.

2.1.1 DEB 패키지 구조

DEB 패키지[8,9]는 데비안 계열 리눅스 운영체제에서 소프트웨어를 배포하기 위해 사용되는 소프트웨어 패키지 포맷이다. 사용자가 패키지를 설치하기 위해 콘솔에서는 dpkg, apt와 같은 도구들도 지원하며 시냅틱 관리자와 같은 GUI 도구들도 제공된다. 패키지 파일은 개별적으로 다운로드 받아 설치할 수도 있으며 네트워크 저장소에 있는 패키지를 다운로드 받아 설치하거나 직접 다운로드 받아 설치 할 수도 있다.

DEB 패키지를 제작하기 위해서는 패키징 과정을 기입한 명세 파일을 작성한 후 dh_make 도구를 사용하여 소스트리와 설정 파일을 하나의 그림의 구조의 형태로 만든다. 다음으로 fakeroot 도구를 사용하여 해당 패키지의 컴파일 단계를 거치면 DEB 패키지 파일이 생성된다.

최종적으로 생성된 DEB 파일은 설정 파일들과 패키지의 바이너리 파일 그리고 소스파일이 단순하게 묶여있는 형태인데 ar -x 패키지명.deb 명령어로 풀어보면 그림과 같은 파일들이 내부에 포함된 것을 확인할 수 있다.

control
md5sums
postinst
prerm
Execution file (x86, arm...)
copyright
changelog
README

그림 1 DEB의 패키지

DEB 패키지의 파일명은 다음과 같은 구조로 되어 있다. 이름 다음에는 버전 정보가 명시되며 그 다음으로 해당 패키지가 지원하고 있는 CPU를 명시하고 있다. 그림과 같이 arm을 명시하고 있다면 패키지 파일 안에는 arm 실행 바이너리가 있다는 것을 의미한다. CPU 부분이 all, any로 되어 있는 경우는 스크립트 파일로 구성된 패키지와 같이 CPU에 독립적으로 작동되는 패키지를 의미한다.

패키지 이름 버전 CPU
package name-1.0.0-arm.deb

그림 2 DEB 패키지의 명명 규칙

2.2 LLVM(Low Level Virtual Machine)

LLVM[1]은 컴파일러를 구성하는 각 모듈들이 잘 분리되어 있어 컴파일, 링크, 런타임 등 각 시점에서 컴파일 최적화 작업을 수행할 수 있는 컴파일러이다. 특히 JVM 처럼 가상머신 기반의 바이너리 코드를 생성할 수 있기 때문에 특정 CPU에 독립적인 코드의 생성이 가능하다. 하지만 JVM이 Java언어만을 지원하는데 비해 LLVM은 C, C++언어를 지원하고 있기 때문에 기존 C, C++로 작성된 많은 소프트웨어에 적용할 수 있다. 또한 x86, x86_64, PPC, ARM, MPIS 그리고 C 소스 Back-end등 다양한 Back-end를 지원하고 있다.

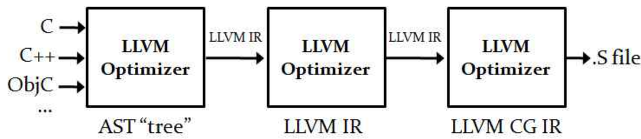

그림 3 LLVM의 컴파일 과정

LLVM은 C, C++, Obj C등의Front-end단의 처리를 위해 GCC를 사용하지만 RTL을 중간 결과물로 내보내는 대신 LLVM 가상머신의 규격대로 작성된 LLVM IR을 생성하는 것이 특징이다. 이 파일은 LLVM의 규격에 맞게 작성된 것으로 특정 CPU와는 독립적으로 작성된다. 그리고 마지막으로 시스템 CPU에 맞는 어셈블리어를 생성해주면 이후 실행파일 생성과정은 기존의 도구들을 사용하게 된다. 즉 LLVM의 핵심적인 기능은 GCC의 front-end에서 처리한 결과를 LLVM 가상머신을 위한 LLVM IR로 변환해주는 것이며 이와 같은 특징은 CPU 독립적인 패키지를 제작하는데 기반 기술이 된다.

2.2.1 LLC

LLC는 실제 LLVM IR인 Bitcode 파일을 다양한 CPU의 어셈블리 언어로 변환 작업을 담당한다. 그림4와 같이 Bitcode를 입력으로 받아 옵션으로 "-march=CPU 타입"과 같이 CPU 타입을 설정해주면 변환 결과로 요청한 CPU 타입의 어셈블리 파일을 생성하게 된다.

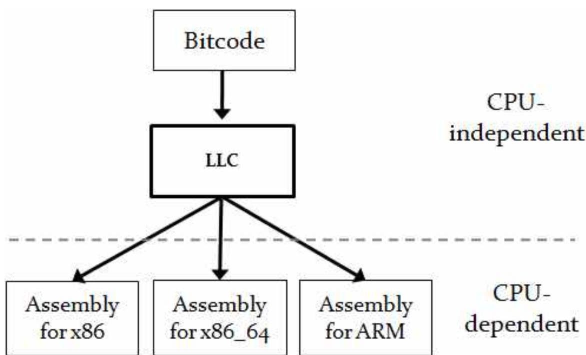

그림 4 LLC의 수행 구조

3. 시스템 설계 및 구현

본 장에서는 상세한 시스템 설계 구조에 대해 설명하고 구현 과정에 대해서 기술한다. 그림5는 제안하는 시스템에서 CPU 독립적인 패키지를 제작할 때 처리되는 각 단계를 보여주고 있다.

각 과정을 간략하게 살펴보면 가장먼저 C, C++ 소스코드를 OceanVM 컴파일러를 사용하여 CPU 독립적인 Bitcode 형태로 변환한다. Bitcode는 Front-end에서 처리해야할 사항들이 완료된 파일이며 LLVM이 이해할 수 있는 명령어로 작성되어 있다.

이렇게 생성된 Bitcode 파일은 DEB 패키지 파일 형태로 패키징 되며 사용자는 네트워크상에 존재하는 패키지 저장소를 사용하거나 해당 패키지를 직접 다운로드 받아 설치를 수행한다.

DEB 패키지에 포함된 Bitcode는 설치 시점에 해당 CPU에 해당하는 어셈블리어 파일로 변환되며 시스템에 설치된 어셈블러와 링커를 이용하여 실행파일을 생성하게 된다. 그림5에서 상단 부분은 패키지 빌드 단계를 말하며 하단 부분은 패키지 설치 단계를 나타내며 자세한 분석은 3.2절과 3.3절에서 다루도록 하겠다.

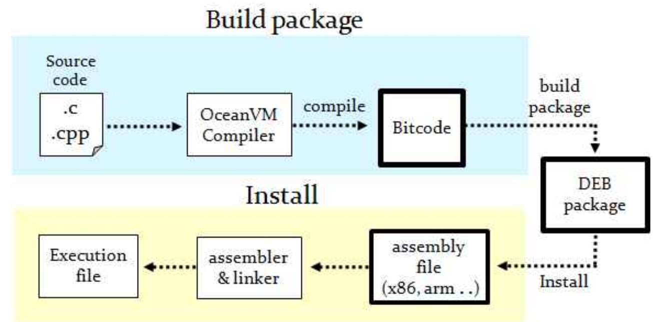

그림 5 패키지의 빌드 과정 및 설치 과정

3.1 OceanVM

OceanVM[6]은 KIST에서 개발하고 있는 가상머신으로 LLVM을 기반으로 제작되어 CPU 호환성 및 운영체제의 호환성을 제공할 수 있는 기술을 개발하였고 기존의 가상머신과는 달리 고성능을 목표로 설계하여 개발되고 있는 중이다. 기존 가상머신들이 작은 코드 사이즈와 완벽한 메모리 보호에 중점을 둔 반면 OceanVM은 수행 중 메모리 사용량과 직접 제어, 그리고 수행성능을 향상 시키는 것에 초점을 둔 것이 특징이다. 현재 데모 어플리케이션들을 운영할 수 있을 정도의 컴파일러 기술과 윈도우즈 호환기술, 그리고 COD센터 등의 개발이 진행된 상태이다.

본 논문은 구체적으로 OceanVM을 리눅스 패키지에 적용하여 CPU 호환성을 보장하는 패키지를 개발한 프로젝트이다. 이를 위해 그림5와 같이 OceanVM 컴파일러가 핵심적으로 사용이 되는 것을 확인할 수가 있다.

3.2 패키지 빌드 단계

리눅스 소프트웨어를 CPU 독립적인 패키지 형태로 빌드하는 단계를 말한다. 기존 패키지 빌드 과정과 비교하여 가장 큰 차이점으로는 패키지 안에 포함된 특정 CPU에 맞게 컴파일 된 실행 바이너리가 CPU 독립적인 Bitcode형태로 교체되었다는 것이다. 기존 패키지들은 설치될 시스템의 CPU 종류에 따라 컴파일 되었기 때문에 하나의 CPU만을 지원할 수 있었다. 하지만 제안하는 시스템에서는 가상머신의 명령어로 작성되어 CPU에 의존적이지 않은 Bitcode를 포함하기 때문에 다양한 시스템에 적용될 수 있다. 빌드를 하기위해 소스코드는 기존 패키지의 소스를 그대로 사용하며, 실제 빌드작업을 수행하는 Makefile에서 컴파일러와 링커를 OceanVM을 이용하도록 변경하는 것이다.

표1은 리눅스에서 많이 쓰이고 있는 압축파일인 'tar' 패키지의 Makefile 이다. CPU 독립적인 패키지를 위한 Bitcode를 생성하기 위해서 수정되는 부분들은 LINK를 llvm-ld로 교체하고 CFLAGS에 -emit-llvm을 추가하여 출력을 Bitcode로 생성하게 한다. 그리고 컴파일러는 llvm-gcc가 사용되어야 하는데 ./configure 하는 단계에서 CC를 llvm-gcc로 설정해준다.

표2 tar 패키지의 Makefile

```
158 #LINK = $(CCLD) $(AM_CFLAGS) $(CFLAGS) $(AM_LDFLAGS) $(LDFLAGS) -o $@
159 LINK = llvm-ld $(AM_LDFLAGS) $(LDFLAGS) -o $@
                    . . . . .
186 #CFLAGS = @CFLAGS@
187 CFLAGS = @CFLAGS@ -emit-llvm
```

빌드를 하면 소스코드의 Front-end의 작업을 수행하고 컴파일 되어진 여러개의 Bitcode 파일들과 Static Library들을 llvm-ld를 이용하여 하나의 Bitcode 파일로 묶어준다.

이렇게 생성된 Bitcode 파일은 DEB 패키지로 제작을 하는데 그림6과 같은 구조로 생성 된다. 기존 DEB 패키지와 차이점은 실행 바이너리 위치에 Bitcode가 대체되어 진다는 것이며 패키지가 설치되고 난 후 실행되는 postinst 스크립트에 Bitcode변환 및 실행파일 생성을 위한 처리부가 추가된다는 것이다.

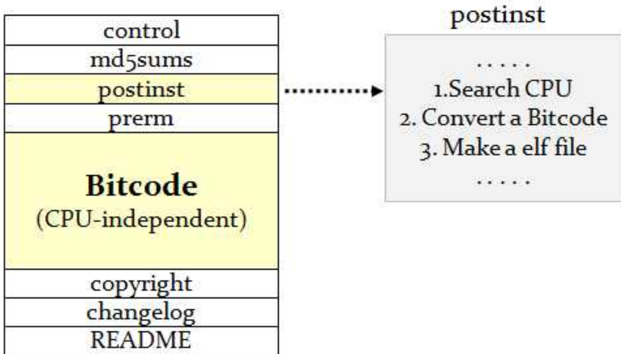

그림 6 CPU 독립적인 DEB 패키지의 구조

생성된 DEB 패키지의 파일명은 특정 CPU에 종속적이지 않기 때문에 CPU를 명시하는 부분을 'all'로 명시하였으며 아래의 같은 형태를 갖는다.

package_name-1.0-all.deb

3.3 패키지 설치 단계

CPU에 비의존적으로 설치할 수 있는 DEB 패키지들은 기존과 동일한 방법으로 네트워크에 위치한 패키지 저장소를 이용하거나 패키지를 직접 다운로드를 받아 패키지 매니저를 이용하여 설치 할 수 있다. 사용자는 패키지를 설치하기 위하여 추가적인 학습이나 명령어를 사용하지 않아도 되기 때문에 기존과 동일한 방법으로 설치가 가능하다.

실제 변환 과정과 실행파일을 생성을 담당하는 파일은 표2의 postinst 스크립트 파일로 패키지의 설치가 종료되는 시점에 수행되어 Bitcode의 변환과정과 실행파일 생성 작업을 추가 수행하게 된다.

설치 과정에 일어나는 수행되는 내용들을 정리하면 다음과 같다.

1. DEB 패키지 파일을 apt-get 등의 패키지 관리 도구를 이용하여 설치
2. 설치가 완료되면 CPU에 독립적인 비트코드가 시스템에 설치
3. 패키지 설치가 완료되면 postinst 스크립트 실행
4. postinst 스크립트는 시스템의 CPU를 검색한 후 비트코드를 어셈블리 파일로 변환
5. 어셈블리 파일은 시스템에 설치된 어셈블러(gas)를 이용하여 오브젝트 파일로 변환
6. 오브젝트 파일은 링커(ld)를 사용하여 실행파일로 변환

표3 postinst 스크립트

```
#!/bin/sh

arch_type=$(uname -m)
x86_compile()
{
  # 어셈블러 코드 생성
  llc /usr/bin/package.bc -march=x86 -o package.s
  # 실행파일 생성
  gcc package.s -o package -ldl -lrt
}
arm_compile()
{
  llc /usr/bin/package.bc -march=arm -o package.s
  gcc package.s -march=armv6 -o package -ldl -lrt
}
                . . . . .
if [ $(expr match $arch_type "arm") = "3" ]
then
  arch_type="arm"
fi

# 시스템의 CPU 판별
case $arch_type in
  i386 | i486 | i586 | i686 ) x86_compile;;
  arm ) arm_compile;;
  ppc ) ppc_compile;;
esac
exit 0
```

패키지 설치를 완료하면 기존의 패키지들과 동일한 방법으로 소프트웨어 패키지를 실행할 수 있다. 실행파일은 설치 시스템의 CPU 명령어로 작성된 실행 바이너리이기 때문에 JVM과 같은 가상머신이 없이도 동작이 가능하다. 따라서 기존 가상머신 기반의 언어로 작성된 소

프트웨어와 비교하여 성능면에서 월등히 뛰어나며 GCC로 컴파일 한 실행 바이너리와 비교해도 거의 동일하거나 조금 나은 수준의 성능을 보여주고 있다.[3]

하지만 설치 시점에 시스템의 CPU 종류에 따라 Bitcode의 변환과정과 실행파일 생성 작업 시간이 추가되는데 이에 대한 자세한 실험결과는 4장에서 다루도록 하겠다.

표3은 설치 시점에 OceanVM을 사용하여 제작된 CPU 독립적인 패키지들의 설치를 위하여 필요한 도구들의 목록을 보여주고 있다. 총 3개의 도구가 필요하며 패키지가 사용하는 동적 라이브러리의 경우 해당 시스템에 미리 설치되어 있다고 가정한다. 다음의 도구들은 반드시 설치 시스템에 설치가 되어야 하는데 패키지 파일의 의존성 부분에 도구들을 의존성 목록에 추가하여 패키지 설치 전에 도구들의 설치가 이루어지게 하였다.

표4 패키지 설치에 필요한 도구들의 크기

이름	설명	x86(Kb)	ARM(Kb)
llc	Bitcode변환	11,462,168	10,142,976
gas	어셈블러	293,652	446,214
ld	링커	445,332	446,408

4. 실험 결과

구현된 시스템의 성능 평가를 위해 리눅스에서 많이 쓰이는 도구들을 선정하여 CPU 독립적인 패키지를 제작하였고, 패키지 빌드시 각 컴파일 단계에서 발생되는 소요 시간과 설치 시점에서 소요된 시간을 측정 하였다.

실험환경으로는 x86의 경우 Pentium Dual Core E5200(2.5Gz) / 4Gb RAM / Ubuntu 9.10 리눅스 배포판이 설치된 시스템을 이용하였다. ARM 시스템은 Openmoko[10]라는 오픈소스 리눅스 모바일 단말기를 사용하였으며 상세 스펙으로는 ARM920T(400Mhz) CPU와 Openmoko용 데비안 배포판을 사용하여 실험을 진행하였다. 컴파일러는 OceanVM(LLVM 2.5 버전 기반)과 GCC 4.3 버전을 사용하여 CPU 별로 패키지 빌드 시간을 측정하였다.

실험은 각 컴파일 단계를 나누어 소요된 시간을 측정하였는데, GCC를 사용한 경우 Makefile을 실행하여 소스코드로부터 실행파일이 생성된 시간(.c->elf)을 측정하였다. LLVM을 사용했을 경우는 소스코드를 비트코드로 변환(.c->.bc) 시간과 비트코드를 특정 CPU의 어셈블러로 변환(.bc->.s)하는 시간, 그리고 실행파일 생성(.s->.elf) 시간으로 나누어 측정하였다.

그림7,8과 같이 x86 CPU와 ARM CPU 모두 소스코드에서 실행파일 생성까지의 컴파일 수행시간만 단순 비교하면 OceanVM을 사용한 경우의 빌드 속도가 더 빨라진 것을 확인할 수 있다. 하지만 OceanVM을 이용한 경우 설치 시점에 Bitcode를 실행 파일로의 변환해야 하므로, 실질적으로 사용자가 체감하는 속도는 변환과정에 소요되는 시간만큼 기존 시스템에 비해 추가적인 시간이 소요된다. 하지만 설치가 완료되면 기존 네이티브 코드와 비슷한 성능으로 프로그램을 수행할 수 있으며, 변환과정은 설치 시점에 한번만 이뤄지는 것이므로 심각한 오버헤드 요소가 되지는 않을 것이다.

추가적으로 사용자 시점에서 DEB 패키지를 설치할 때 소요된 시간을 x86 시스템에서 측정한 결과 wget 패키지의 경우 x86시스템에서 일반 DEB 패키지를 설치한 경우 2.221초가 걸린 반면 OceanVM을 사용한 패키지의 경우 5.22초가 걸렸다. 즉 추가적으로 발생된 약 3초의 시간은 그림7의 wget 패키지에서 Bitcode변환시간(.bc->.s)과 실행파일 생성시간(.s->.elf)이 사용자가 설치하는 시점에 추가적으로 발생되었다는 것을 확인할 수 있다.

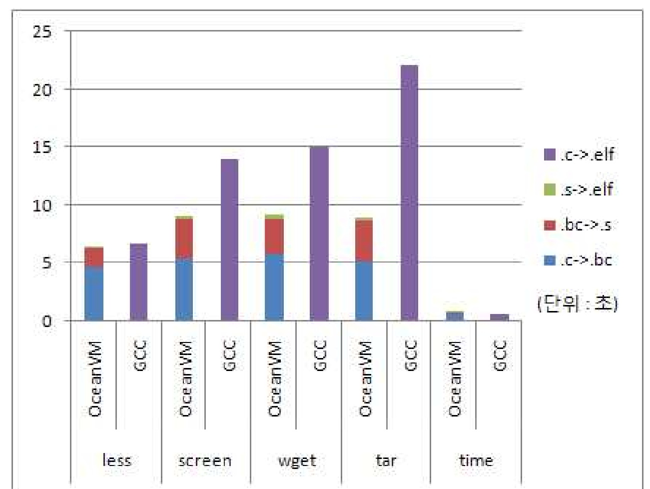

그림 7 x86 CPU에서의 패키지 빌드 시간

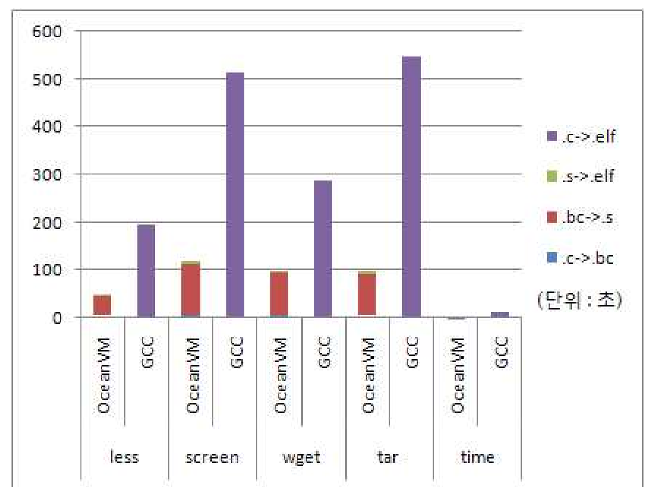

그림 8 ARM CPU에서의 패키지 빌드 시간

5. 논의

실험결과를 통해 제안하고 있는 시스템의 가능성과 성능 평가를 통해 추후 발전시켜야 할 사항들에 대해 정리하면, CPU 독립적인 패키지를 위하여 제안된 시스템은

CPU에 상관없이 패키지 설치를 가능하게 해주지만 설치 시점에 발생하는 변환 과정에 소요되는 시간과 설치에 여러 도구들이 필요하다는 단점이 존재하였다. 장, 단점에 대해 구체적으로 정리해보면 아래와 같다.

CPU 독립적으로 패키지를 배포하여 얻을 수 있는 장점으로 임베디드 장비와 같이 기기에서 x86용으로만 제공되었던 많은 패키지들을 포팅 하던 노력을 줄여줄 수 있었다. 그리고 사용자는 자신이 사용하려는 기기의 CPU를 알지 못해도 x86 환경과 동일한 패키지들을 설치하여 사용 가능하였다. 향후 PC 성능과 견줄 수 있는 ARM CPU 기반의 넷북이나 스마트 폰 등이 활성화 될 것으로 예상되며, x86 기반에서 사용되었던 많은 소프트웨어들이 넷북이나 스마트 폰에서 사용률이 증가될 것으로 예상된다. 한번의 패키징으로 다양한 CPU를 지원해 줄 수 있기 때문에 개발자나 사용자 모두 사용 편리성을 제공받을 수 있을 것이다.

하지만 설치 시점에 변화과정에 추가적인 시간이 소요되는 단점이 있다. 일반적으로 패키지를 설치할 때 걸리는 시간에 추가적으로 변환과정에 소요되는 시간이 사용자에게 큰 불편함을 주지는 않는다고 판단되며 설치 시점에 딱 한번 소요되는 비용이므로 크지 않을 거라고 예상된다. 그리고 다른 단점으로는 변환과정에 필요한 도구들을 사전에 설치하고 있어야만 한다는 것이다. 이는 임베디드 기기와 같이 적은 저장 공간을 갖는 시스템의 경우 부담이 될 수 있는 요소이다.

6. 결론

본 연구로 향후 ARM용 리눅스 기반의 넷북, 스마트폰과 같은 많은 모바일 기기들을 대상으로 CPU 독립적인 패키지를 제작 기술을 확보할 수 있었다. 하지만 연구 결과에서 보았듯이 사용자 측에서 추가되는 설치시간이 소요된다는 것과 변환 도구들이 사전에 설치되어야 한다는 단점들을 앞으로 해결해야 할 것이다.

앞으로 진행해야 할 연구 목표로 공식 OceanVM 패키지 저장소를 제공하여 일반 사용자들이 사용할 수 있도록 제공 할 것이다. 그러기 위해서는 많은 패키지들에 대해 작업이 요구되며 안정성 및 버그 검출 등의 검증 작업이 필요하다. 그리고 개발자들이 개발한 소프트웨어를 본 시스템을 이용하여 패키지를 제작할 경우 가이드라인을 제공할 수 있도록 작업과정과 패키지 포맷에 대해 표준화 작업이 이루어 질 예정이다.

7. 참고문헌

[1] Chris Lattner, "Introduction to the LLVM Compiler Infrastructure", 2006 Itanium Conference and Expo, San Jose, California Apr. 2006.
[2] Chris Latter, "M.S. Theis, LLVML An Infrastructure for Multi-Stage Optimization", Computer Science Dept., University of Illinois at Urbana-Chanpaign, Dec. 2002.
[3] 김재진, 이석영, 김수현, 문수묵, "임베디드 시스템에서 LLVM과 GCC의 성능 및 최적화 분석", 차세대 컴퓨터 2009 추계 학술대회, 2009. 11.
[4] J. Smith and R Nair, "Virtual Machines", Elsevier, 2005
[5] Latter, Chris, Vikram Adve, "Architecture for a Next-Generation GCC", May. 2003
[6] OceanVM Site, http://www.oceanvm.org
[7] LLVM Site, http://www.llvm.org
[8] http://www.debian.org
[9] http://en.wikipedia.org/wiki/Deb_(file_format)
[10] http://wiki.openmoko.org/wiki/Neo_FreeRunner
[11] www.winehq.org