가상 머신간 성능 비교

김재진[1], 정동헌[1], 김수현[2], 문수묵[1]
서울대학교 전기컴퓨터공학부[1], 한국과학기술연구원[2]
kjj7999@altair.snu.ac.kr, clamp@altair.snu.ac.kr,
suhyunk@gmail.com, smoon@snu.ac.kr

A Performance Comparison of Various Virtual Machines

Jaejin Kim, Dong-Heon Jung Suhyun Kim, Soo-Mook Moon

요 약

가상 머신은 중간코드로 컴파일 되어 한 프로그램을 여러 플랫폼에서 수행 가능하게 한다. 이러한 가상 머신에는 이미 널리 알려진 썬 마이크로시스템즈의 자바 가상 머신과 구글의 안드로이드 플랫폼의 달빅 가상 머신 그리고 애플이 지원하는 LLVM 등이 있으며, 파이어폭스의 TraceMonkey, 구글 크롬의 v8, 사파리의 SquirrelFish Extreme 같은 자바스크립트 엔진도 일종의 가상 머신으로 볼 수 있다. 가상 머신은 필연적으로 성능 저하를 동반하게 되는데, 이는 가상 머신의 주요 이슈 중 하나이다. 본 논문에서는 간단한 벤치마크를 통해서 이들 가상 머신간의 성능을 비교하고, 각 가상 머신의 특징을 알아본다. LLVM은 여러 단계에 걸친 컴파일 전략으로 JIT 컴파일을 사용하였을 때 높은 성능을 보이나 JIT 컴파일을 사용하지 않았을 경우는 매우 낮은 성능을 보인다. 달빅 가상 머신은 인터프리터 모드에서 자바 가상 머신 보다 조금 나았으나, 아직 개발된 지 얼마 되지 않아 JIT 컴파일러가 없다는 것이 약점이다. 자바스크립트 엔진들은 동적 언어인 자바스크립트를 지원하는 특성상 최적화를 적용하지 못해 비교적 낮은 성능을 보였다.

1. 서 론

가상 머신(Virtual Machine, VM)은 가상의 하드웨어를 소프트웨어 적으로 구현하여, 그 위에서 프로그램이 실행되도록 하는 것이다.[1] 가장 널리 알려진 가상 머신으로는 썬 마이크로시스템즈에서 개발한 자바 가상 머신(JVM)이 있으며, 마이크로소프트에서 개발된 공통 언어 런타임(CLR)도 널리 알려진 가상 머신이고, 파이어폭스의 TraceMonkey, 크롬의 v8 같은 자바스크립트 엔진들도 일종의 자바스크립트(JavaScript) 가상 머신으로 볼 수 있다.

자바 프로그램은 자바 가상 머신에 의해 운영체제나 하드웨어에 독립적인 중간 언어(Intermediate Language)인 바이트코드로 번역되어 자바 가상 머신이 설치되어 있는 환경이면 어디서나 실행할 수 있다. 이런 플랫폼 독립성은 가상 머신의 가장 큰 장점이자 특징이다. 자바 가상 머신은 그 외에도 쓰레기 수집기(Garbage Collection)과 메모리 보호를 제공해 바이트코드 실행 시 안정성을 높여준다. [2]

- 본 연구는 지식경제부 및 한국산업기술평가관리원의 산업원천기술개발사업(정보통신)의 일환으로 수행하였음. [KI002119, 고성능 가상머신 규격 및 기술개발

최근 개인용 컴퓨터 외에 소프트웨어를 구동할 다양한 플랫폼이 주목 받고 있다. 특히 아이폰의 등장과 함께 스마트폰이 유망한 모바일 플랫폼으로 떠오르고 있다. 스마트폰에는 범용 운영체제가 설치되어 있어, 같은 운영체제가 설치된 폰에는 같은 프로그램을 실행시킬 수 있게 되었고, 1GHz CPU 가 탑재되는 등 모바일 플랫폼의 하드웨어 자원도 풍부해짐에 따라 모바일 플랫폼에서 소프트웨어인 어플리케이션이 중요하게 취급 받게 되었다. 애플의 앱스토어에는 이미 10만건이 넘는 어플리케이션들이 올라와 있고, 구글도 안드로이드 마켓을 열고, SK텔레콤이나 KT에서도 앱스토어를 여는 등 어플리케이션을 중요하게 생각한다.

또한 TV쪽에서는 IPTV (Internet Protocol Television) 가 나와 기존의 컨텐츠를 일방적으로 받아 보던 때와는 달리 인터넷을 이용해 양방향 텔레비전 서비스를 이용하게 되었다. 이처럼 다양한 플랫폼이 늘어나고 발전함에 따라 플랫폼 독립적인 가상 머신의 필요성이 커지고 있다. 개발자가 가상 머신에 기반하여 프로그램을 작성하면 여러 플랫폼에서 그 프로그램을 실행 시킬 수 있다는 장점이 있고, 가상 머신 위에서 작동하기에 그 호환성을 보장 받을 수 있다.

그리고 인터넷 기반의 클라우드 컴퓨팅 기술에서

소프트웨어를 하나의 서비스로 사용하도록 하는 것, 즉 SaaS(Software as a Service) 는 중요한 개념이며, 클라우드 컴퓨팅을 통해 제공되는 어플리케이션이 다양한 플랫폼에 동작하게 하기 위해서는 성능 좋은 가상 머신이 필요하다. [3]

하지만 가상머신은 프로그램 수행이 실제 머신이 아니라 그 중의 일부를 할당 받은 가상 머신 상에서 이루어지는 것으로 프로그램 코드를 실행시간에 해석해야 하거나 언어가 지원하는 기능을 만족시키기 위한 여러 부담으로 인한 필연적인 성능 저하를 수반하게 된다. 또한 가상머신은 각각 자기의 주된 목적에 맞게 설계 되어있기 때문에 이에 따른 성능 차가 매우 상이하다. 내장형 플랫폼의 하드웨어가 많이 발전했다고는 하나, 그만큼 어플리케이션도 고도화되어 복잡해지고 용량이 커짐에 따라 내장형 플랫폼에서의 가상 머신의 성능은 아직도 중요한 문제이다. 그래서 이 논문에서는 여러 주요한 가상 머신의 성능 비교를 통해 각 가상 머신들의 특징을 살펴볼 것이다.

본 논문의 구성은 다음과 같다. 2장에서는 비교할 다양한 가상 머신들을 소개하고, 3장에서는 가상머신의 성능 비교를 위해 수행할 실험환경을 제시하고, 4장에서는 실험을 통해 나온 결과를 제시하고, 5장에서 결론을 맺도록 한다.

2. VM 소개

이 절에서는 실험을 수행할 다양한 가상 머신을 소개한다.

2.1 Low-Level Virtual Machine (LLVM)

Low-Level Virtual Machine 은 2000년 일리노이 대학에서 시작된 프로젝트이며, 컴파일 타임, 링크 타임, 런타임 등 여러 시점에서의 최적화에 중점을 둔 가상 머신이고, 2005년부터는 애플에서 지원 중이다.[4,5] LLVM-GCC 라 불리는 GCC 기반의 C/C++ 전단부를 제공하고 있으며, Objective C, Fortran, Ada 등의 언어를 변환할 수 있는 전단부도 있다.

LLVM은 자체적인 바이트코드를 가지고 있으며, 이 중간 코드(Intermediate representation)는 고수준 언어에 독립적인 명령어 집합(Instruction set)과 자료형 체계(type system)을 가지고 있다. SSA(Static Single Assignment) 로 작성된 3-어드레스 코드 형태로 되어있다.[6]

이 중간코드는 머신 독립 적인 최적화를 수행할 수 있고, LLVM 후단부를 통해 타깃 머신의 어셈블리로 변환될 수 있다. 또한 LLVM은 중간 코드를 바로 인터프리터 형태로 실행할 수 있는 도구도 제공한다.[7]

2.2 자바 가상 머신 (JVM)

자바 가상 머신(JVM)은 1996년 썬 마이크로시스템즈 사가 발표하였고, 규격화된 자바 바이트코드 (Java bytecode)를 수행할 수 있는 환경이다. 자바 언어(Java language) 로 작성된 프로그램은 운영체제나 하드웨어에 독립적인 중간 언어인 자바 바이트코드로 번역되어 자바 가상 머신 위에서 동작한다.[13]

현재 가장 많이 쓰이는 가상 머신 규격으로 윈도우즈, 리눅스, 맥 OS X 등 대부분의 운영체제와 다양한 하드웨어 플랫폼에 이식(porting) 되어 사용되고 있다.

기본적으로 스택 기반의 자바 바이트코드를 하나씩 차례대로 실행하는 인터프리터(interpreter) 방식으로 수행되며, 내장형 시스템에서는 대개 인터프리터 모드로 실행된다. 필요에 따라 Just-In-Time compile (JITC)과 Ahead-Of-Time compile (AOTC)등의 기법을 통해 바이트코드를 미리 머신 코드로 변환하여 수행함으로써 수행시간을 단축 시키고 있다.

2.3 달빅 가상 머신 (Dalvik VM)

달빅 가상 머신(Dalvik VM)은 구글(google)에 의해 개발된 가상 머신으로 안드로이드(Android) 플랫폼에 포함된 가상 머신이다.

달빅 가상 머신(Dalvik VM)은 메모리와 프로세서 속도의 제약이 있는 시스템에 적합한 Dalvik Executable(.dex) 포맷으로 변환된 프로그램을 실행한다. dx 라 불리는 툴을 사용해 자바 class 파일을 dex 포맷의 파일로 변환할 수 있으며, 여러 클래스들이 하나의 dex 파일에 포함될 수 있다. dex 파일 포맷은 독립적인 명령어 체계(Instruction set)을 가지고 있으며, 일반적으로 같은 class 파일들이 압축된 jar 보다 크기가 작다.[9]

스택기반의 자바 가상 머신(JVM)과는 다르게 레지스터 기반의 가상 머신이다. 일반적으로 레지스터 기반의 머신은 스택 기반의 머신에 비해 코드 크기는 더 커지나 수행시간은 감소한다. 현재 스마트폰 등의 모바일 시스템은 과거에 비해 메모리가 비교적 여유가 있고, 성능이 중요시 되고 있어 레지스터 기반의 머신은 효율적이다. [15]

2.4 TraceMonkey

TraceMonkey는 모질라 파이어폭스(Mozilla FireFox) 3.1 브라우저에 내장된 자바스크립트용 가상 머신 엔진이다. TraceMonkey는 기본적으로 자바스크립트 소스 코드를 중간코드인 LIR(Low-level Intermediate Representation)로 변환한 뒤 명령어를 하나씩 수행하는 인터프리터 방식을 사용한다.

TraceMonkey 역시 빠른 성능을 위해 자바 가상 머신(JVM)에서와 같은 Just-in-time compilation을 지원한다. 하지만 동적 언어 (dynamic language)라는 자바스크립트의 특성상 자바 가상 머신과 같이 메소드(method)단위로의 컴파일이 어렵기 때문에 코드의 제어 흐름(control flow)상의 자주 사용되는 경로(Hot pass)를 단위로 컴파일 하는 트레이스 단위 컴파일(traced-based compilation)방식을 채택하여 사용하고

있다.[10]

2.5 SquirrelFish Extreme

SquirrelFish Extreme (SFX)는 웹킷(Webkit)의 자바스크립트 엔진이며, 사파리(Safari) 4.0 브라우저에 탑재되어 있다. [11]

SFX는 TraceMonkey와 마찬가지로 중간코드 인터프리터의 형태로 중간코드를 실행하던 기존의 SquirrelFish 엔진에서 발전된 형태이다. SFX는 기존의 SF와 달리 컨텍스트 쓰레딩(context threading)이라는 기법을 이용해 컴파일 된 코드를 만들어내는 full JIT 방식을 이용하여 수행한다.

2.6 V8

V8 은 구글(Google)에 의해 개발된 오픈 소스 자바스크립트 엔진이며, 구글 크롬(Google Chrome) 브라우저에 탑재되어 있다. [12]

V8은 바이트코드를 인터프리터 형태로 실행되지 않고 수행하기 전에 고유의 머신 코드(native machine code) 로 컴파일 하여 수행하여 성능을 올린다. 그리고 inline caching 등의 최적화 기법을 사용하여 성능을 올린다.

3. 실험 환경

이 절에서는 VM 비교를 위해 수행할 실험 환경을 제시한다.

실험은 Intel Pentium 4 2.8GHz CPU, 메인 메모리 2G 리눅스(Linux-2.6.28) 머신에서 수행하였고, 각 가상 머신의 버전은 LLVM (v2.6), J2SE (v1.6), Dalvik VM, TraceMonkey (v1.7), SFX(Revision 48534), v8을 사용하였고, JVM, LLVM, TraceMonkey 같은 경우는 JIT을 켜고 끌 수 있으므로, JIT을 사용한 경우와 사용하지 않은 경우로 나눠서 비교하였다.

LLVM 은 C/C++ 언어로 작성된 프로그램을 수행하는 가상머신이고, 자바 가상 머신이나 달빅 가상 머신은 자바 언어로 작성된 프로그램을 수행할 수 있고, TraceMonkey, SFX, v8 같은 자바스크립트 엔진들은 말 그대로 자바스크립트로 작성된 프로그램을 수행한다. 이처럼 각각의 가상 머신들이 변환하여 수행할 수 있는 고수준 언어(High-Level Language)의 종류가 다르기 때문에, 각 가상 머신 별로 다른 언어로 작성된 벤치마크를 사용해야 하였다. 작성된 언어의 특성에도 영향을 받기에 벤치마크의 선정에 주의가 필요했다.

언어의 차이로 인한 문법 차이를 최소화하기 위하여 다른 벤치마크들에 비하여 비교적 간단한 연산을 반복적으로 수행하는 Pendragon Software Corporation의 CaffeineMark 벤치마크를 사용하였다. CaffeineMark는 내장형 시스템의 성능 측정에 자주 쓰이는 벤치마크로, 원래 자바로 작성되어 있어, 실험을 위해 C와 자바스크립트 버전의 CaffeineMark 벤치마크를 만들었다. 아래 그림들은 CaffeineMark의 각 언어에 따른 실제 구현 코드의 일부이다. 이 코드는 Loop 벤치마크의 주요 루프 중 하나로 약간의 언어 규약상의 차이를 제외하고는 거의 동일한 코드임을 할 수 있다.

```
for(int k = 1; k < FIBCOUNT; k++)
{
        int l = FIBCOUNT + dummy;
        j1 += l;
        k1 += 2;
        if(fibs[k - 1] < fibs[k])
        {
                int i1 = fibs[k - 1];
                fibs[k - 1] = fibs[k];
                fibs[k] = i1;
        }
}
```

그림 1. CaffeineMark for Java

```
for (k = 1; k < 64; k++)
{
        int l = 64 + dummy;
        j1 += 1;
        k1 += 2;
        if(fibs[k - 1] < fibs[k])
        {
                int i1 = fibs[k - 1];
                fibs[k - 1] = fibs[k];
                fibs[k] = i1;
        }
}
```

그림 2. CaffeineMark for C

```
for (k = 1; k < 64; k++)
{
        var l = 64 + dummy;
        j1 += 1;
        k1 += 2;
        if(fibs[k - 1] < fibs[k])
        {
                var i1 = fibs[k - 1];
                fibs[k - 1] = fibs[k];
                fibs[k] = i1;
        }
}
```

그림 3. CaffeineMark for JavaScript

JIT(Just-in-Time) 컴파일은 가상 머신에서 런타임에 프로그램을 실행하기 전에 바이트코드(bytecode)를 원시 코드(native code)로 변환해 주는 기법으로, 성능을 크게

올려준다. 자바 가상 머신, LLVM, TraceMonkey, v8, SFX 등은 JIT 컴파일을 지원하고, 자바 가상 머신, LLVM, TraceMonkey는 실행 시 JIT 옵션을 통해 JIT 컴파일 사용 유무를 조절 할 수 있다. 하지만 달빅 가상 머신은 아직 JIT 컴파일을 지원하지 않는다. 이 JIT 컴파일의 영향에 대해서도 성능을 측정해보기로 한다.

4. 실험 결과

실험은 우선 JIT 컴파일을 사용하여 가장 많이 쓰이는 자바 가상 머신을 기준으로 하여 LLVM, TraceMonkey, SFX, v8과 비교하였다. 실험 결과는 아래와 같다.

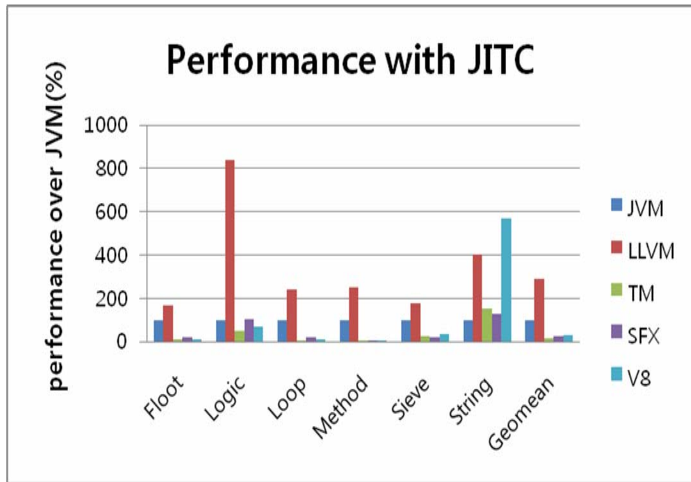

그림 4. JIT을 사용하였을 때 VM 성능비교

LLVM 은 자바 가상 머신에 비해 3배 가까이 우수한 성능을 보였으며, 실험한 가상 머신 중 가장 성능이 좋았다. 현재 LLVM 은 레지스터 기반(Register base)의 중간코드(Intermediate code)를 통해 GCC와 거의 동일한 수준의 머신 코드를 생성하고 있기 때문으로 보인다.

다만, 여기서 고려해야 하는 것은 CaffeineMark 벤치마크의 특성인데 CaffeineMark는 간단한 계산 등의 루프를 반복적으로 수행하는 형태의 벤치마크이기 때문에 작은 부분을 한번만 JIT 컴파일을 하고 나면 더 이상 중간코드가 돌 필요가 거의 없다. 그렇기 때문에 JIT 컴파일의 효과가 인터프리터에 비해 매우 커진다.

안드로이드에 포함된 달빅 가상 머신(Dalvik VM)의 경우 안드로이드 플랫폼 자체가 아직 초기 단계이기 때문에 공식적인 JIT 컴파일러가 구현되어 있지 않아서 비교를 해볼 수 없었다.

TraceMonkey와 SFX, V8등 자바스크립트 엔진들은 아직 개발 초기단계로 성능 면에서는 아직 걸음마 단계이다. (10~30%) 현재 많은 오픈 소스 진영에서 3최적화가 동시다발적으로 진행 중이다. 동적 언어(Dynamic Language)인 자바스크립트는 동적 타입(dynamic type), 클로저(closure)등을 사용하기 때문에 수행시간 중 동적으로 처리를 해야 한다는 근원적인 특성과 코드를 미리 알기 힘든 웹 환경의 현실적인 특성이 성능 최적화에 발목을 잡고 있는 상황이다.

그리고 자바스크립트의 경우 동적 언어(dynamic language)로서 수행도중 변수의 타입(type)들을 결정하기 때문에 미리 중간코드를 변환하는 것은 거의 불가능하여 이 부분을 분리하는 것이 불가능하다. 그렇기 때문에 자바스크립트 엔진인 SFX와 TraceMonkey, V8의 경우 수행시간에 자바스크립트를 중간코드로 바꾸는 전단부(front-end)의 수행시간이 포함되어있기 때문에 완전히 공평한 비교라고 보긴 힘들다. 하지만 CaffeineMark의 경우 100 줄 내외의 비교적 간단한 코드이기 때문에 중간코드로 변환하는 시간은 실제 벤치마크 수행시간에 비하면 작을 것으로 예상된다.

자바 가상 머신(JVM)의 경우 초창기 매우 느린 수행성능을 가지고 있었고, 버전이 높아지면서 JITC 등의 최적화 기술이 발달하여 현재는 native 컴파일러의 수준에 근접해 가고 있다. 하지만 이것은 PC 성능의 자원(resource)을 가진 플랫폼에서의 성능이고 가상머신이 주로 사용되는 내장형 시스템에서는 JIT 컴파일이 제한 받거나 전혀 JIT 컴파일 과정에서의 오버헤드가 문제가 되는 경우도 많다. 그렇기 때문에 실제 내장형 시스템에서의 성능을 확인하기 위해서 JIT 옵션을 끄고 인터프리터(interpreter) 모드로 돌린 실험이 필요하다.

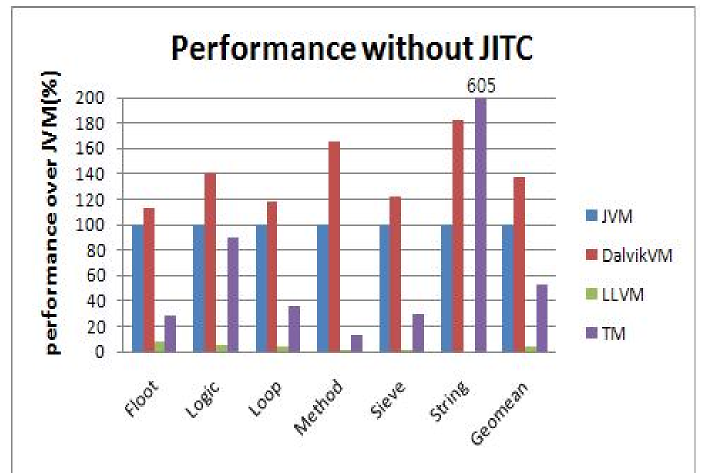

그림 5. 인터프리터 수행시 성능 비교

인터프리터 모드로 실험한 결과는 그림 5와 같다. y축은 자바 가상 머신(JVM)에서 인터프리터 모드로 실행한 결과를 기준(100%)으로 잡은 성능 값이다. SFX나 V8의 경우 Full JIT으로 실행하기 때문에 인터프리터 실험에서 제외하였고 LLVM에서 String 벤치마크는 돌지 않아서 실험에서 제외하였다.

우선 JIT 컴파일을 하지 않은 JVM 인터프리터는 아래 표와 같이 JIT 컴파일을 사용한 것에 비해 크게 낮은 성능을 보였다. CaffeineMark 벤치 마크 자체가 작은 중심 메소드 하나를 반복하는 것이라 JIT 컴파일의 효과가 매우 뛰어나기 때문에 JIT 컴파일에 비해 그것을 사용하지 않은 인터프리터에서의 결과가 더욱 낮게

나온 것으로 보였다.

	JVM	Dalvik	LLVM	TM	SFX	V8
noJIT	8.2%	11.3%	0.3%	4.3%		
JIT	100%		290.4%	13.6%	25.6%	30.9%

표 1. JVM-JIT 기준 JITC 사용 여부에 따른 성능 비

달빅 가상 머신의 경우 대체적으로 다른 가상 머신에 비해 자바 가상 머신과 성능이 비슷하였으나, 평균적으로 자바 가상 머신의 1.37배 더 좋은 성능을 보였다. 달빅 가상 머신의 바이트코드는 자바 가상 머신의 바이트코드와 거의 유사하였으나 스택의 인자를 피연산자(operand)로 사용하는 자바와는 달리 레지스터를 피연산자 (operand)로 사용하게 되어있고, 몇몇 바이트코드가 좀더 세분화 되어있었다. [13, 14]

Java bytecode	Dalvik bytecode	Java bytecode	Dalvik bytecode
iadd	add-int	newarray	new-array
isub	sub-int	athrow	throw
imul	mul-int	goto	goto/16
fcmpl	cmpl-float	if_icmpeq	if-eq
fcmpg	fcmg-float	if_icmpne	if-ne
getfield	iget	putstatic	sput
getfield	iget-byte	putstatic	sput-byte

그림 6. Java 바이트코드와 Dalvik 바이트코드 (일부)

레지스터 기반의 머신은 스택기반의 머신에 비해 명령어에 피연산자(operand)가 포함되어 있어 코드 크기(code size)는 늘어나게 되나 아래 표와 같이 같은 하이 레벨 코드(code)를 나타내는데 더 적은 명령어 (instruction)가 필요해 명령어를 디스패치하는 비용이 줄어들게 된다. 한 연구에서는 레지스터 기반의 머신이 스택 기반의 머신보다 수행시간이 25%~30% 정도 짧아진다고 밝혔고, 달빅 가상 머신의 성능 향상은 이 범주 안이다. 따라서 달빅 가상 머신의 성능 향상은 레지스터 기반 머신이기 때문으로 보인다.[15]

code	stack machine (JVM)	register machine (dalvik VM)
a=b+c	iload c iload b iadd istore a	add-int a, b, c

표 2. 스택 머신과 레지스터 머신의 비교

LLVM의 인터프리터 성능이 낮은 것이 눈에 띈다. LLVM 인터프리터는 모든 벤치마크에서 JVM 인터프리터의 8% 이하의 성능을 보여주었으며, 평균 3.5% 밖에 되지 않았다.

LLVM은 여러 단계에서 최적화를 통해 타깃 머신의 어셈블리 코드를 만들어내는 컴파일 전략으로 처음 제안되었으며 플랫폼 독립성과 함께 쓰레기 수집기(Garbage Collector)를 통해 바이트코드 실행의 안정성에도 중점을 두고 있는 자바 가상 머신과는 설계 철학이 다르다. 때문에, LLVM은 일반적으로 중간 코드를 다시 컴파일 하여 타깃 머신의 어셈블리 코드를 얻는데 중점을 두고 설계 되었고, 인터프리터는 단순하게 설계 되어서 미흡한 점이 많았다.

TraceMonkey는 String 벤치 마크에서 높은 성능을 보였지만, 역시 동적 언어인 자바스크립트를 위해 만들어진 한계상 다른 벤치마크에서는 자바 가상 머신에 비해 평균 30%의 성능 밖에 나오지 않았다.

String 벤치 마크에서 JIC 컴파일을 사용하였을 때 v8의 성능이 뛰어났고, 다른 자바스크립트 엔진들도 자바 가상 머신보다 뛰어난 성능을 보였다. JIT 컴파일을 사용하지 않고 인터프리터 모드로 실행시켰을 때도 TraceMonkey이 다른 벤치마크들 보다 성능이 크게 뛰어났다. 이와 같이 String 벤치마크에서는 자바스크립트 엔진들이 좋은 모습을 보였다. String 벤치마크는 라이브러리를 호출하여 문자열을 연결하거나 문자열에서 특정한 문자열을 찾거나 하는 동장을 반복하는데, 자바스크립트의 문자열 관련 라이브러리가 잘 되어있어서 더 나은 성능이 나온 것으로 보인다.

5. 결 론

이번 실험을 통해 여러 가상머신의 성능을 비교해 보았다. 자바 가상 머신(JVM)은 스택 기반의 가상 머신으로 중간코드를 실행하기 때문에 코드 크기 등에서 여러 이점을 가지고 있고, 쓰레기 수집기를 사용하여 안정성에서는 뛰어나지만 성능에서는 인터프리터에서는 달빅 가상 머신에 비해 부족한 모습을 보였고, JIT 컴파일을 사용할 경우는 LLVM에 비해 부족한 모습을 보였다.

달빅 가상 머신은 인터프리터 모드에서 전반적으로 자바 가상 머신 보다 뛰어난 성능을 보였으며 이는 스택 머신인 자바 가상 머신과는 달리 레지스터 기반의 바이트코드를 사용하기 때문으로 보인다. 하지만 달빅 가상 머신이 설치된 안드로이드 플랫폼 자체가 개발 된지 얼마 안된 플랫폼이라 JIT 컴파일러가 구현되지 않은 것이 약점이다.

LLVM은 GCC기반의 프론트엔드와 레지스터 기반의 중간코드를 사용하고 최적화를 많이 수행하기에 간단한 코드들을 JIT 컴파일할 시에는 우수한 성능을 보이지만 인터프리터 성능은 다른 VM들에 비해 매우 떨어졌다. LLVM은 일반적으로 중간코드를 어셈블리로 바꾸어서 실행하거나 JIT 컴파일을 사용하여 실행하게 되어 있고, 인터프리터는 미흡한 점이 많았다.

자바스크립트(JavaScript) 언어를 타깃으로 하는 SFX와 TraceMonkey, V8은 개발자 편의를 고려한 자바스크립트(JavaScript) 언어 특성상 이를 지원하는 가상 머신 엔진은 성능상의 큰 손실이 있었다.

그리고 모든 가상 머신에서 JIT 컴파일을 수행 하였을 때 성능이 크게 증가함을 보았을 때, 가상 머신의 성능을 올리기 위해서는 JIT이 필수적임을 알 수 있다.

Reference

[1] J. Smith and R. Nair, Virtual Machines, Elsevier, 2005

[2] J. Gosling, B. Joy, and G. Steele, The Java Language Specification Reading, Addison-Wesley,1996.

[3] Luis M. Vaquero et al., A Break in the Clouds: Toward a Cloud Definition, ACM SIGCOMM, Volume 39, Issue 1 January 2009, Pages 50–55

[4]LLVM Home, http://www.llvm.org

[5]Chris Lattner, M.S. Thesis, LLVM: An Infrastructure for Multi-Stage Optimization Computer Science Dept., University of Illinois at Urbana-Champaign, Dec. 2002.

[6] Vikram Adve et al., LLVA: A Low-level Virtual Instruction Set Architecture, MICRO-3, SanDiego,CA,Dec.2003.

[7] Chris Lattner, Introduction to the LLVM Compiler Infrastructure, 2006 Itanium Conference and Expo, San Jose, California, Apr. 2006.

[8] The Java Virtual Machine Specification, 2nd edition, Tim Lindholm, Frank Yellin, Prentice Hall, 1 Jul 1999

[9] Dalvik VM Internals sites.google.com/site/io/dalvik-vm-internals

[10] Mozilla Wiki, wiki.mozilla.org/JavaScript:TraceMonkey

[11] SquirrelFish Project, trac.webkit.org/wiki/SquirrelFish

[12] V8 JavaScript Engine, code.google.com/p/v8/

[13] JAVA Virtual Machine, O'REILLY, Jon Meyer & Troy Downing

[14] Bytecode for the Dalvik VM, Google Inc. 2007

[15] Virtual Machine Showdown: Stack vs. Register Machine, Yunhe Shi, David Gregg, Andrew Beatty, M. Anton Ertle, VEE'05