임베디드 시스템의 자바 가속 기술

Java Acceleration Technology on Embedded System

박준석(J.S. Park) 휴대클라이언트연구팀 선임연구원
김명규(M.G. Kim) 휴대클라이언트연구팀 선임연구원
한동원(D.W. Han) 휴대클라이언트연구팀 책임연구원, 팀장

자바 애플리케이션의 이식성을 보장해주는 'WORA' 모델을 실현하기 위해서 바이트코드에 기반한 자바는 바이트코드 인터프리터를 포함하는 구조적 한계로 인해 성능상의 문제를 갖고 있다. 최근에 서버에서 정보가전에 이르기까지 자바 기술을 확산시키기 위해 자바는 J2EE, J2SE, J2ME의 3영역으로 나누어지고 셀룰러폰, PDA 등 스마트 핸드헬드 기기에는 J2ME 환경이 제공되고 있다. 데스크톱 PC의 고성능화와 다양한 가속 기술의 개발로 인해 성능 문제가 보완되어 수많은 자바 애플리케이션이 데스크톱 PC에서 개발되어 왔으나 CPU, 메모리, 전력 등 자원 제약적 특성을 갖는 임베디드 시스템은 데스크톱 PC에 적용된 자바의 성능 향상 기술을 적용하기에 부적절하여 이에 적합한 새로운 자바 가속 기술이 개발되고 있다. 본 고에서는 임베디드 시스템에서 자바의 성능 향상을 위해 개발된 자바 가속 기술을 소프트웨어 및 하드웨어 측면에서 살펴보고 대표적인 상용 기술에 대해 고찰하였다.

I. 서 론

하드웨어 플랫폼에 독립적으로 자바 애플리케이션의 이식성을 보장해주는 자바의 'WORA(write-once, run-everywhere)' 모델은 바이트 코드 구조를 제시하는 구조적 문제로 인해 성능상의 문제를 갖고 있지만 데스크톱 PC의 고성능화로 성능 문제가 보완됨으로써 수많은 자바 애플리케이션이 데스크톱 PC에서 개발되어져 왔다. 최근에 서버급에서 정보가전기기에 이르기까지 자바 기술을 확산시키는 노력의 일환으로 자바 런타임 환경은 J2EE, J2SE, J2ME의 3가지 영역으로 나누어지고 셀룰러폰, PDA, 웹 패드, 웹 TV, 셋톱 박스 등의 스마트 핸드헬드 기기와 정보가전 기기들에는 J2ME 자바 런타임 환경이 제공되고 있다. 그러나 임베디드 기기는 기존의 데스크톱 PC와는 달리 CPU, 메모리, 전력 등 시스템 자원과 비용 측면에서 많은 제한을 갖고 있기 때문에 기존의 데스크톱 PC에서 개발된 자바 가속 기술을 임베디드 기기에 적용하는 것은 여러 가지 문제점을 갖는다. 임베디드 시스템에 자바를 통합하여 자바의 처리 속도를 향상시키기 위해서는 비용, 전력 소모, 메모리, 공간 등 다양한 제약 요소를 고려하여 비용을 최소화 하면서 성능을 향상시키는 것이 중요하다.

최근 무선 인터넷의 활성화와 더불어 자바 기반의 무선 인터넷 플랫폼이 등장함에 따라 자바 애플리케이션의 보급이 증가하고 있으나 이러한 자원 제약 시스템은 자바 애플리케이션의 성능 문제가 부각됨으로써 이에 적합한 새로운 자바 가속 기술들이 개발되고 있다. 현재까지 임베디드 시스템을 위한 자바 가속 기술은 자바 가상 머신의 최적화, JIT (Just-in-Time) 컴파일러 등 소프트웨어 기술과 자

바 전용 프로세서 및 코프로세서 등 하드웨어 기술이 개발되었으나 이들 기술은 임베디드 시스템이 갖고 있는 자원 제약 요소의 일부분만 해결하는 기술적 한계를 갖고 있다.

본 고에서는 임베디드 시스템에서 자바 실행 환경을 위한 J2ME 기술과 자바 성능 향상을 위해서 고려해야 할 요소, 그리고 현재까지 자바의 성능을 향상시키기 위해서 개발된 자바 가속 기술을 소프트웨어 및 하드웨어 측면에서 살펴보고 대표적인 상용 기술에 대해 고찰하였다.

이를 위해 I장 서론에 이어 II장에서는 임베디드 시스템의 자바에 관하여 기술하고, III장에서는 임베디드 시스템에서 자바의 문제점과 자바 가속 기술에 대해 살펴보았다. IV장에서는 상용 기술을 살펴보고 V장에서 결론을 맺는다.

II. 임베디드 시스템의 자바

1. J2ME의 배경

임베디드 시스템 및 정보 가전 등 다양한 종류의 소형 정보 기기들이 등장하고 이러한 기기들은 모두 인터넷에 연결될 것으로 예상됨에 따라 썬 마이크로시스템즈(사)는 한 가지 버전의 자바로서는 모든 정보 기기를 지원할 수 없다는 판단 아래 자바 런타임 플랫폼을 컴퓨팅 환경에 따라 3가지 버전 즉 J2EE, J2SE, J2ME으로 분할하였다. J2EE는 엔터프라이즈 버전으로 서버 급의 컴퓨터를 목적으로 하고, J2SE는 데스크톱 PC급의 컴퓨터를 목적으로한 자바 런타임 플랫폼이다. J2ME는 페이저, 셀룰러폰, 스크린폰, 디지털 셋톱박스, 카네비게이션 시스템을 포함한 광범위한 일반 소비자 기기를 목적으로 설정한 자바 런타임 환경이다.

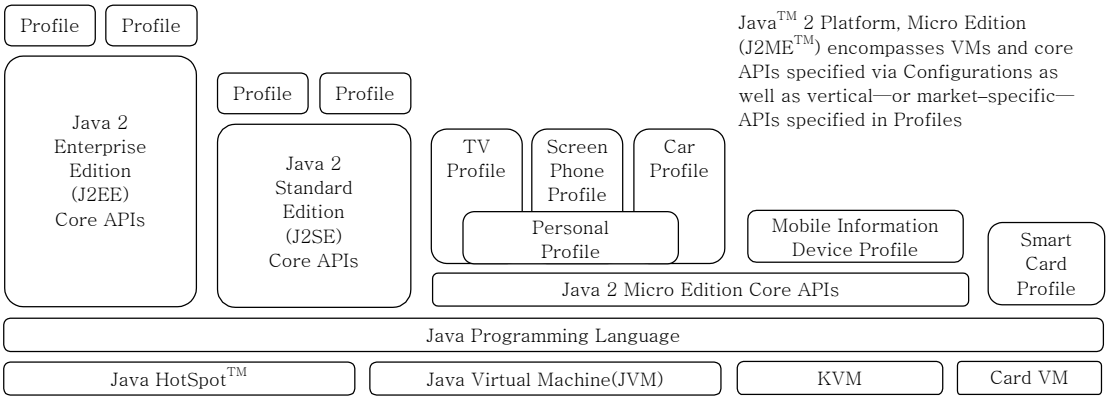
J2ME는 자바 런타임 환경을 다바이스에 커스터마이징하기 위해서 컨피규레이션 및 프로파일을 사용한다. 컨피규레이션은 특정 타입의 디바이스에서 가동되는 코어 클래스의 집합과 특정 자바 가상 머신의 런타임 환경으로 정의된다. 프로파일은 비슷한 컴퓨팅 환경을 갖는 기기들을 그룹화하여 이들 도메인에 특정한 클래스를 J2ME 컨피규레이션에 추가하여 애플리케이션을 정의한다.

2. J2ME의 구성

(그림 1)은 여러 가지 자바 가상 머신, 컨피규레이션 및 프로파일 간의 관계를 정의한다. 일반적으로 J2SE의 자바 가상 머신은 JVM으로서 총칭하고, J2ME의 자바 가상 머신은 KVM과 CVM 등으로 구분하며 이들은 모두 JVM의 서브세트이다.

가. 컨피규레이션

컨피규레이션은 특정 타입의 디바이스에서 동작



(그림 1) 자바 가상 머신의 분류

하는 코어 클래스의 집합과 특정 자바 가상 머신으로서 자바 런타임 환경을 정의하는데 현재까지 2개의 컨피규레이션이 정의되었다.

1) CLDC

CLDC(Connected Limited Device Configuration)는 한정된 규격을 갖는 디바이스에서 J2ME의 구현에 필요한 가장 기본적인 라이브러리 세트와 자바 가상 머신을 정의한다. CLDC는 9.6kbytes 이하의 느린 네트워크 연결, 배터리 공급의 제한된 전원, 128KB 이상의 ROM, 32KB 이상의 RAM를 갖는 디바이스를 목적으로 한 규격으로서 KVM과 함께 160kbytes의 메모리와 16비트 프로세서를 지원하는 낮은 시스템 사양을 갖는 단말기를 목적으로 한 버전이다. 따라서 이러한 단말기에 불필요하다고 판단되는 실수 연산, 리플렉션, 쓰레드 그룹, 위크 레퍼런스, 사용자 정의 클래스 로더, 직렬화, 원격 메소드 호출 등의 자바 언어 기능은 배제하였다. 이는 범용의 자바 애플릿을 수용하기 보다는 단말기의 제약 사항을 수용하여 자바 프로그램을 개발하도록 함으로써 자바의 이식성을 상실한 것이다. CLDC는 J2SE의 java.io, java.lang, java.util 패키지의 서브세트로 구성되며 Javax.microedition 패키지를 추가로 포함하고 있고 상위에 2개의 프로파일, 즉 PDAP(Personal Digital Assistant Profile)와 MIDP(Mobile Information Device Profile)을 정의한다.

2) CDC

CDC(Connected Device Configuratuion)는 J2SE의 스트립 다운 버전이며 CLDC 클래스들이 포함되어 있다. 따라서 CDC는 CLDC의 상위에 만들어졌으며 CLDC 디바이스 용으로 개발한 애플리케이션 또한 CDC 디바이스에서 실행된다. CDC는 CVM과 함께 사용되며 자바 플랫폼을 위해 2Mbytes 이상의 메모리와 32비트 프로세서를 갖고 네트워크에 연결되는 단말을 목적으로 한 자바 버전이다. 이 버전을 수용한 단말기는 데스크톱 PC에서와 같이 범용의 자바 애플릿을 다운로드 받아 실행할 수 있다.

<표 1> CDC와 CLDC의 비교

J2ME 범주	CDC	CLDC
가상머신	CVM	KVM
프로세서	32bit	16~32bit
메모리 요구사항	2~16Mbytes	160~512Kbytes
제약사항	Awt 에 종속적인 GUI 제거	실수연산, Finalization 메소드, RMI 등을 지원하지 않으며 제한된 오류 처리를 제공
타깃 애플리케이션	셋톱박스, 고급 PDA	휴대폰, 저급 PDA

CDC가 지원하는 디바이스는 홈 게이트웨이, 스마트폰, PDA, 오거나이저, 홈 어플라이언스, 차량 항법 시스템 등을 포함한다. CDC는 J2SE API의 서브세트로서 실수연산, 클래스로더 클래스, 네이티브 프로세스, 멀티쓰레드, 직렬화 클래스, 리플렉션 API, 파일 시스템, J2SE 스타일 네트워크, HttpConnection 인터페이스가 추가된 javax.microedition.io 패키지, 자바 시큐리티 등을 지원하고 상위에 파운데이션 프로파일을 지원한다.

나. 프로파일

1) MIDP

이 프로파일은 CLDC를 기반으로 설계된 자바 클래스 라이브러리를 정의하며 96×54 픽셀 스크린, 1비트의 모노크롬 디스플레이, 키패드/키보드/터치스크린 입력장치, 136K의 비휘발성 메모리 등의 규격을 갖는 디바이스 용이다. MIDP에서는 간단한 레코드 관리, 기본적인 UI를 제공하고 HTTP 1.1 프로토콜의 서브세트를 지원한다.

2) 파운데이션 프로파일

이 프로파일은 CDC와 함께 일반 소비자 기기와 임베디드 시스템을 겨냥하여 J2ME 자바 런타임 환경을 제공하기 위한 API 집합이다. CDC에 있는 CVM은 파운데이션 프로파일(foundation profile) 라이브러리를 위한 엔진에 해당된다. 파운데이션 프로파일은 J2SE v1.3을 기반으로 하고 있지만 GUI

를 제공하는 Java.awt 패키지를 포함하고 있지 않아 J2SE의 기본 패키지에서 java.awt에 의존하는 부분들은 포함하지 않는다.

III. 자바 가속 기술

자바는 바이트코드 인터프리터 문제로 인해 매우 느린 것으로 인식되어 왔으나 썬에서 자바 바이트코드를 마이크로프로세서 명령으로 컴파일하는 JIT 컴파일레이션 기술을 개발함으로써 속도 문제를 어느 정도 해결해 주었다. 그러나 이러한 솔루션은 자원이 풍부한 데스크톱과 서버에서는 우수한 성능을 제공해 주지만 램(RAM)과 가상 메모리 등의 무한 메모리를 사용하는 비용을 요구하므로 정보 가전이나 임베디드 시스템과 같이 제한된 자원을 갖는 시스템에 자바 가상 머신을 통합하는 것은 시스템이 갖고 있는 여러 가지 한계 요소를 고려해야 한다.

첫째 요소는 성능으로서 자바는 원래 바이트코드가 머신 코드로 해석되는 단계를 거쳐 실행되므로 일반적으로 느린 단점을 갖고 있다. 이러한 단점을 보완하기 위해서 썬 마이크로시스템즈(사)는 자바 바이트코드를 순수 머신 코드로 컴파일하는 JIT 컴파일 기술을 데스크톱과 서버 시스템에 도입하여 속도 문제를 해결하였지만 이 기술은 램과 가상 메모리 등 무한 메모리를 사용해야 하는 부담을 제공한다.

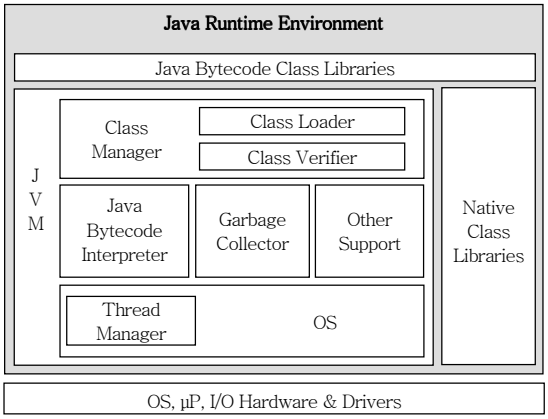
두번째 문제는 런타임 시와 프로그램 저장을 위해서 필요한 메모리다. 자원 제약성을 갖는 시스템에서 메모리는 아주 중요한 자원이므로 수백 메가의 메모리와 무한대의 가상 메모리를 갖는 데스크톱 PC와 서버에서 구현된 자바 가상 머신을 정보가전이나 임베디드 기기 규모에 맞도록 줄이는 것은 매우 어렵다. 메모리의 효율성을 고려하여 작성한 프로그램일지라도 실제 가용한 메모리보다 많은 런타임 메모리를 요구할 수 있고, 요구한 메모리 크기를 성공적으로 할당 받았는지 체크하지 않는 경우 시스템 오류를 야기할 수도 있다.

자바 가속 기술을 개발할 때에는 이러한 사항들의 상호 연관성을 고려하여 trade-off를 결정하는 것이 중요하다. 본 장에서는 성능에 영향을 주는 자바 구성 성분과 자바 가속 기술에 대해서 살펴 본다.

1. 성능에 영향을 미치는 자바 성분

(그림 2)는 자바 런타임 환경의 구성도를 나타내며 자바의 구성 성분 중 성능에 영향을 미치는 요소는 크게 3가지이다.

현재까지 개발된 자바의 성능 향상 기술은 주로 이들 요소들의 처리 속도를 향상시키 데 초점을 두고 있다.



(그림 2) 자바 런타임 환경의 구성도

가. 자바 바이트코드 인터프리터

자바 바이트코드 인터프리터는 자바 명령어를 마이크로프로세서 명령어로 해석하는 부분으로 자바 가상 머신의 핵심 성분이다. 자바 명령어는 시스템의 마이크로프로세서 명령어로 한 바이트씩 해석되므로 자바 머신 코드의 실행은 자바 가상 머신의 성능 병목현상 중 제일 큰 부분을 차지한다. 일반적으로 자바 바이트코드 명령어는 다수의 마이크로프로세서 명령어로 해석되며 자바 가상 머신과 이에 대응하는 명령어 집합은 스택 기반 구조를 갖는다.

나. 클래스 관리기

클래스 관리기는 어떤 클래스 파일을 언제 로드할 것인지를 결정한다. 클래스 로더는 자바 가상 머신의

시큐리티의 일부분으로 바이러스 등의 악성 코드를 제거하기 위해 클래스 파일과 각 바이트코드를 스캔한다. 클래스 파일이 로드되고 검증이 완료되면 실행을 위해서 바이트 코드가 자바 인터프리터로 전달된다.

다. 유휴 메모리 수집기

유휴 메모리 수집기는 프로그램에서 더 이상 사용하지 않는 메모리를 수집하여 관리하는 기능을 수행한다. 자바 프로그래밍 언어는 프로그래머가 직접 메모리 자원을 관리할 필요가 없는 유휴 메모리 수집 메커니즘을 갖는 객체 지향 언어이다. 따라서 객체가 생성되어 초기화됨에 따라 실행시 메모리를 사용하고, 사용중인 메모리는 더 이상 객체가 필요하지 않다는 사실을 유휴 메모리 수집 프로세스가 결정할 때 비로소 해제된다. 이러한 유휴 메모리 수집 프로세스는 자바 가상 머신이 자바 응용 소프트웨어를 실행하는 동안 병렬로 실행되기 때문에 런타임 성능에 많은 영향을 미친다.

2. 자바의 성능 문제

자바 소프트웨어의 성능을 향상시키기 위해서는 성능상의 이슈가 되는 문제점을 이해하는 것이 필요하다. 본 절에서는 성능상 이슈가 되는 자바의 특성을 살펴보았다.

가. 자바 바이트코드 사용

자바를 사용하여 개발된 소프트웨어는 직접 마이크로프로세서의 머신 코드로 컴파일되는 것이 아니라 자바 가상 머신의 머신 코드를 구성하는 자바 바이트코드 명령어로 해석되는데 이러한 자바 바이트코드 명령어는 두 가지 속성을 갖고 있다. 첫째는 자바 바이트코드는 특정 마이크로프로세서에 종속되지 않는다는 것이고, 다른 하나는 한 개의 바이트코드 명령어는 다수의 마이크로프로세서 명령으로 번역되는 high-level 오퍼레이션을 수행한다는 점이다. 이러한 속성은 실행 속도를 저하시키고 전원 사용을 증가시키는 단점을 제공하므로 배터리에 의해 전원이 공급되는 기기와 적은 메모리, 저속의 마이크로프로세서를 갖는 자원 제약 디바이스에서 특히 문제가 된다.

나. 중간 런타임 플랫폼

자바 응용프로그램을 자바 바이트코드로 컴파일한다는 것은 프로그램을 한 시스템에 특정하게 하는 것이 아니라 자바 런타임 환경이 이식된 모든 플랫폼에서 실행할 수 있음을 의미한다. 이것은 바이트코드를 시스템의 중간 언어로 만들고, 자바 런타임 환경을 중간 런타임 플랫폼으로 만든다. J2ME와 같은 중간 런타임 플랫폼은 마이크로프로세서와 운영체제로 구성되는 하부 시스템이 자바 런타임 플랫폼과 자바 응용 소프트웨어를 동시에 실행해야 하므로 순수하게 컴파일된 소프트웨어와 비교할 때 성능상의 단점을 갖는다. 또한 자바 플랫폼 그 자체는 자바 소프트웨어를 실행하기 위해서 마이크로프로세서를 심하게 이용할 뿐만 아니라 메모리 풋프린트와 런타임을 위해서 추가적인 메모리를 필요로 한다. 따라서 메모리 요구사항을 최소화하면서 동시에 자바 소프트웨어의 실행 성능을 향상시킬 필요가 있다. 자바 소프트웨어의 바이너리 코드는 자바 가상 머신에 의해 실행되며 자바 가상 머신은 각 자바 바이트코드 명령어를 순수 마이크로 프로세서 명령으로 인터프리트하므로 시스템은 자바 프로그램을 실행하기 위해서 자바 가상 머신과 자바 소프트웨어를 병렬로 실행한다.

다. 스택 기반 구조

휴대형 정보기기에서 사용되는 상업용 마이크로프로세서는 레지스터 기반 구조를 갖는데 반해 자바 가상 머신과 이에 대응하는 머신 코드는 스택 기반 구조를 갖는다. 자바 가상 머신이 스택 기반 구조를 갖는다는 것은 자바 바이트코드 명령어의 실행이 상업용 마이크로프로세서의 동작 방법과는 상당히 다르다는 것을 의미한다. 이러한 스택 구조 처리 방식은 임시 데이터, 값, 메소드 인수들을 변수와 공통 스택을 통해 전달하고 자바 스택은 시스템의 주 기

억장치에 상주하며 동작하므로 각 스택의 상호작용 시 기억장치로의 접근을 필요로 하므로 성능의 저해 요소로 나타난다. 따라서 스택 기반의 자바 가상 머신과 레지스터 기반의 마이크로 프로세서간의 갭을 해결하기 위해 CPU 내에 변수와 스택 엔트리의 국부화를 꾀함으로서 성능을 향상시킬 수 있다.

3. 소프트웨어 가속 기술

일반적으로 자바의 실행 속도를 높이기 위해 소프트웨어 기술이 널리 사용되어져 왔으나 충분한 메모리 효율성을 제공하도록 최적화된 자바 가상 머신 일지라도 고도의 복잡한 응용을 실행하기 위해서는 고성능 프로세서를 사용하지 않는 한 적절한 성능을 제공하지 못하기 때문에 소프트웨어 가속 기술은 임베디드 시스템에는 적합하지 않은 것으로 간주되고 있다.

가. JVM 최적화

자바 가상 머신의 작업 부하를 분석하여 성능에 문제를 야기하는 병목 부분을 부분적으로 해결하여 자바 가상 머신의 최적화를 도모한다. 바이트코드 해석 루프는 어셈블러로 처리하고 공통 데이터에 대해서는 CPU 레지스터를 이용한다. 충분한 메모리 성능을 제공하지만 고성능 프로세서를 사용하지 않으면 복잡한 애플리케이션에게 적절한 성능을 제공하지 못한다. 이 방식은 특정 마이크로프로세서의 구조에 종속되는 단점을 가지며 비용과 전원 제약 사항 때문에 임베디드 기기에는 적합하지 않은 방식이다.

나. JIT 컴파일레이션

이 방법은 자바 애플리케이션의 성능을 마이크로프로세서 명령어로 직접 컴파일된 소프트웨어와 동일하게 하기 위해서 컴파일러를 자바 가상 머신에 통합하여 자바 가상 머신이 자바 프로그램을 실행하는 것과 동시에 자바 프로그램의 일부를 컴파일하는 것이다. JIT(Just in Time) 컴파일레이션을 위해서는 2가지 방식이 사용된다. 하나는 프로그램을 구성하는 메소드와 클래스가 프로그램 실행 이전에 로드될 때 컴파일되는 것이고, 두번째 방식은 실행 프로그램을 분석하여 자주 사용하는 클래스, 메소드, 코드 부분을 결정하여 이러한 부분들을 프로세서의 명령어 코드로 컴파일하고 실행한다. 일반적으로 컴파일러 크기는 100Kbytes 이상이며 컴파일된 코드는 6~8배까지 확장되므로 큰 용량의 RAM 캐시를 필요로 한다. 따라서 추가 메모리 자원을 필요로 하므로 메모리 자원이 중요한 임베디드 시스템에는 부적합한 가속 기술이다. 이 방식은 성능상의 문제를 해결하지만 특정 프로세서에 기반한 자바 플랫폼에 종속적일 뿐 아니라 메모리 양을 증가시키고, 전력 소모를 증가시키는 문제를 갖는다.

다. AOT 컴파일레이션

이 방법은 자바 프로그램을 실행하기 전에 미리 완전히 컴파일하는 것이다. AOT(Ahead of Time) 컴파일레이션을 위해 2가지 방법이 사용된다. 하나는 C 나 C++ 프로그램이 컴파일, 저장, 분배되는 방식과 마찬가지로 자바 프로그램을 컴파일, 저장하여 실행 코드를 생성하는 것으로 특정 시스템에 종속되어 이식성을 보장하지 못한다. 또 다른 방식은 프로그램이 표준 자바 클래스 파일로서 배포되며 실행에 앞서 로드되기 이전에 컴파일된다. 이 방식은 실행하기 이전에 다소 긴 지연을 초래하므로 애플릿을 실행하는 데 있어서는 사용할 수 없는 방식이다.

4. 하드웨어 가속 기술

자바의 실행 속도를 개선하기 위해서 자바 가상 머신을 직접 실리콘으로 구현하는 실행 모델이다. 자바의 실행 속도를 높이기 위한 가장 원시적인 하드웨어 기술은 마이크로프로세서의 속도를 증가시키는 것이다. 그러나 스택 기반의 자바 가상 머신은 메모리 기반 스택을 이용하므로 마이크로프로세서와 메모리 사이에 현격한 속도 차이로 인해 마이크로프로세서의 속도를 향상시키는 것이 자바의 성능을 선형으로 증가시키지 않을 뿐만 아니라 시스템 비용과 전력 소모를 증가시킨다. 따라서 임베디드

시스템에서는 바이트코드를 프로세서의 머신 명령어로 변환하는 오버헤드를 감소시키고 뿐만 아니라 자바의 런타임 특성을 제공하는 새로운 방식의 하드웨어 기술이 개발되고 있다. 이들 하드웨어 가속 기술은 스택 처리, 멀티쓰레딩, 가비지 콜렉션, 오브젝트 주소화 및 심볼릭 resolution 등을 지원해 줌으로써 최적화를 통해 범용 프로세서보다 더 좋은 성능을 제공하는 것을 목적으로 한다.

가. 자바 전용 프로세서

자바 바이트코드를 전용 프로세서에서 직접 실행하는 것으로서 성능은 좋지만 통합 및 개발의 복잡성으로 상당한 오버헤드가 존재한다. 기존의 애플리케이션 또는 운영체제를 지원하지 못하므로 항상 다른 프로세서와 함께 동작되어야 한다.

나. 자바 코프로세서

코프로세서는 자바 바이트코드를 기존 코어 프로세서의 명령어로 변환시킨다. 이 방식은 상당한 하드웨어와 소프트웨어의 통합 노력을 필요로 하며 기존 운영체제와 통합하기 어렵다. 또한 하드웨어 게이트에 대한 추가 공간과 동작에 필요한 추가 전원을 필요로 하므로 제조 비용이 비싸다. 코어 프로세서와 느슨하게 연결되어 있으므로 비교적 느리게 실행되는 경향이 있다.

5. 하이브리드 가속 기술

코어 프로세서의 구조를 확장하여 코어 프로세서에서 직접 자바 바이트코드를 가속시키기 위해 하드웨어와 소프트웨어 솔루션이 결합된 형태이다. 이는 기존의 운영체제, 미들웨어, 응용 프로그램들과 함께 자바 바이트코드를 직접 실행할 수 있는 향상된 단일 프로세서 솔루션이다. 프로세서 내부에 명령어 세트를 추가하는 구조 확장 방식은 기존의 구조를 변경할 필요없이 모든 프로세서 자원을 재사용한다. 확장된 코어는 자바와 마이크로프로세서의 머신 코드를 효과적으로 실행하여 자바의 이식성과 성능 향상을 동시에 달성할 수 있게 해주므로 휴대 단말에 적합한 방식으로 간주되고 있다.

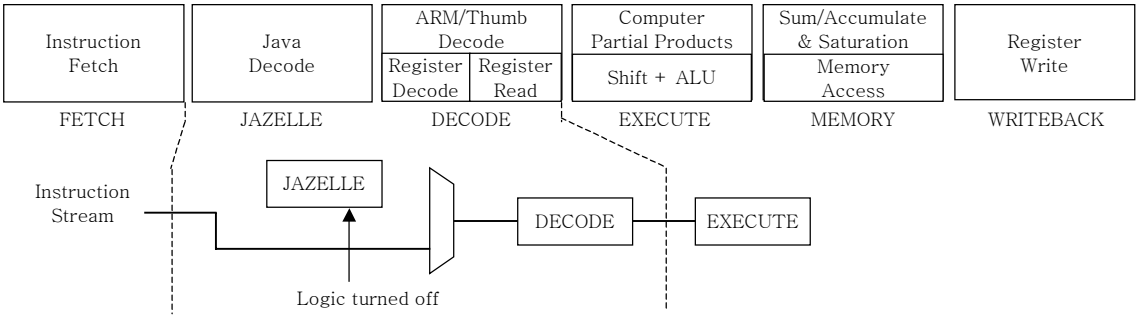
IV. 사례 연구

1. Arm사의 Jazelle 구조

영국의 ARM 사에서는 자바의 실행 속도를 향상시키기 위해 기존의 ARM 프로세서 구조를 확장한 Jazelle을 발표하였다. ARM926EJ로 명명된 ARM 프로세서는 32비트 길이를 갖는 ARM 명령어 세트와 자주 사용되는 명령어를 16비트 길이로 압축한 Thumb 명령어 세트, 그리고 자바 바이트코드 명령어 세트를 포함한다. 이 프로세서는 '자바 상태'와 'ARM/Thumb 상태'의 2가지 상태를 가지며, 실행되는 명령어의 종류에 따라 이 상태를 스위칭하면서 동작한다. 자바 바이트코드 명령어 세트는 '자바 상태'를 생성하며 이 상태에서 프로세서는 자바 바이트코드를 패치, 디코드하여 자바 오퍼랜드 스택을 유지하여 자바 머신처럼 동작한다. (그림 3)은 두 가지 상태에서 명령어 스트림 처리의 파이프라인 구조를 보여준다. 자바 상태로 들어가기 위해서는 신규 ARM 명령어 'Branch-to-Java'를 실행하여 프로세서를 자바 상태로 놓고 지정된 목적지 주소로 점프하여 자바 바이트코드 실행을 개시한다. 자바 상태에 있는 동안에 ARM 프로그램 카운터는 32비트로 확장되어 자바 바이트코드의 주소를 가리키며 자바 바이트코드는 패치되어 두 단계로 디코드된다. 이와는 대조적으로 ARM/Thumb 상태에서는 단일 디코드 단계를 갖는다. 프로세서의 상태는 CPSR (Current Processor Status Register) 비트에 기록되며 CPSR은 인터럽트와 예외처리를 수행할 때 자동으로 저장되고 복구되기 때문에 Jazelle은 운영체제에서 사용하는 기존의 ARM 인터럽트/예외처리 모델과 호환성을 갖는다.

가. 자바 상태

자바 상태에서 프로세서는 일부 ARM 레지스터



(그림 3) 명령어 스트림 파이프라인(5단계: ARM/Thumb 상태, 6단계: Java 상태)

를 자바 머신의 특정 기능(R6=스택 포인터, R0-R3=스택의 상부 요소, R4=로컬 변수)에 할당한다. 이와 같이 ARM 레지스터에 Jazelle 확장에 필요한 모든 상태를 유지하여 하드웨어를 재사용하므로써 자바 머신을 구현하는 데 필요한 추가 로직이 적게 해준다. 실제로 대부분의 애플리케이션 실행시 런타임 스택 길이는 아주 작아 메모리 액세스를 최소로 감소시키므로 ARM 레지스터에 스택의 상부 4개 요소를 저장하는 것은 프로세서의 성능에 중요한 기여를 한다. 스택 오버플로나 언더플로는 하드웨어에 의해 자동으로 처리된다.

나. 자바 바이트코드 분류

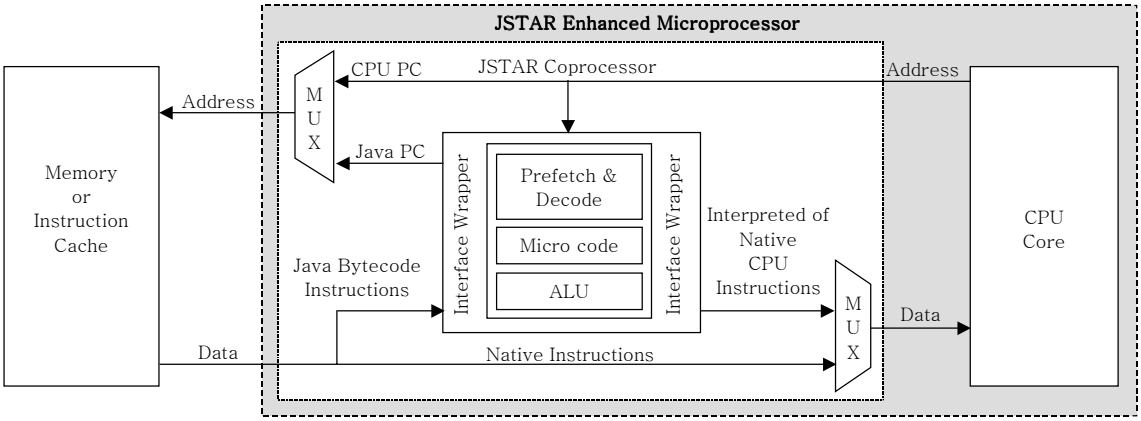
자바 바이트코드는 '직접 실행 코드', '에뮬레이션 코드', '정의 않된 코드'의 3가지로 분류된다. 자바 바이트코드의 대부분은 하드웨어로 직접 실행되고, 에뮬레이션 코드는 ARM 명령어 코드의 최적화된 시퀀스로 에뮬레이트된다. Jazelle 기술은 가상 머신의 인터프리터 루프를 제거하여 이를 VMZ라는 코드로 대체하므로 추가 로직이 필요없게 해준다. 정의 않된 코드는 프로세서가 '자바 상태'를 벗어나 '예외 처리기'로 복귀되도록 야기하는데 이는 자바 바이트코드를 확장하기 위해 사용되는 방식이다.

2. Nazomi의 JSTAR 코프로세서

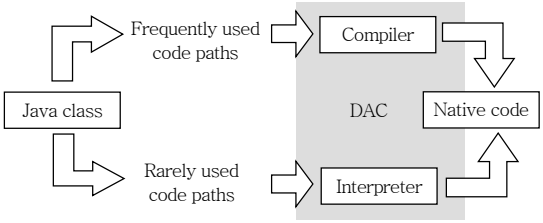
Nazomi사는 자바의 런타임 실행 속도를 향상시키기 위해 마이크로프로세서 확장을 위한 JSTAR 코프로세서, 하드웨어 기반의 JIT 컴파일러 칩인 JA108, 자바 전용 프로세서 등 다양한 하드웨어 칩 기술을 보유하고 있다. 마이크로 프로세서를 확장하는 방법에는 인스트럭션 세트를 확장시키는 방식과 인스트럭션 경로 인터프리터 방식이 있다. 이러한 방법은 모두 마이크로프로세서가 자바 바이트코드 인스트럭션을 인터프리트 하는 것으로서 펜티엄 프로세서의 MMX 확장이 멀티미디어 처리를 가속시켜 주는 것과 비슷한 방식이다. (그림 4)는 JSTAR 인스트럭션 경로 코프로세서의 구조도를 보여준다. Nazomi 사의 JSTAR 자바 코프로세서는 인스트럭션 경로 인터프리터에 해당하는 방식이며 마이크로프로세서 코어와 JSTAR 코프로세서를 통합함으로써 JSTAR 코프로세서는 모든 자바 바이트코드 인스트럭션을 패치하고 최적화된 마이크로프로세서 머신 코드를 생성하여 이를 마이크로프로세서로 전달한다. 따라서 JSTAR 코프로세서는 마이크로프로세서의 정상적인 실행과 특성을 변경시키지 않고 새로운 하드웨어 핀의 추가없이 기존의 운영체제와 도구들을 그대로 사용 가능한 기술이다.

3. Insignia사의 동적 적응형 컴파일러

Insignia사는 임베디드 시스템에 적합한 Jeode 플랫폼에 대한 기술을 보유하고 있다. Jeode 플랫폼의 핵심은 동적 적응형 컴파일러(Dynamic Adaptive Compiler: DAC)이다. DAC 방식은 컴파일러 버퍼와 프로그램 힙(heap) 간의 메모리를 동적으로



(그림 4) JSTAR 인스트럭션 경로 코프로세서의 구조도



(그림 5) Nazomi의 DAC 구조도

교환함으로써 프로그램의 메모리 요구량을 시스템에서 가용한 메모리에 최적으로 부합하도록 프로그램의 동작을 배치시키는 것이다. 인터프리터만 사용하는 자바 가상 머신보다 훨씬 빠르며 작고 configurable 메모리를 사용하여 RAM 오버헤드가 적다.

DAC는 동적으로 로드된 클래스를 지원하며 자바 응용프로그램 쓰레드와 병렬로 실행되는 쓰레드로서 실행된다. 컴파일하는 동안 자바 쓰레드가 실행될 필요가 있다면 즉시 스케줄링되어 인터프리터에 의해 실행된다. DAC는 (그림 5)과 같이 동적 컴파일러와 바이트코드를 인터프리트 하기 위한 인터프리터 둘 다를 포함하며 바이트코드는 최초 로드될 때 인터프리터를 통해 실행된다. 프로파일러가 각 메소드에 대한 런타임의 레코드를 유지하여 메소드가 여러 번 호출되는 것이 발견되면 DAC는 그것을 컴파일하여 최적화시킨다. 따라서 추후 그 메소드에 대한 모든 호출은 컴파일러가 생성한 마이크로프로세서 머신 인스트럭션을 사용한다. 이 방식은 자원 제약 시스템에 적용할 수 있는 컴파일레이션 방식을 제공할 뿐만 아니라 응용 프로그램의 응답성을 향상시킨다.

V. 결 론

자바는 모든 하드웨어 플랫폼 상에 자바 애플리케이션의 이식성을 보장하기 위해서 자바 바이트코드라는 중간 코드를 생성하므로 본질적으로 성능상의 문제를 갖고 있다. 특히 메모리, CPU, 전력 사용 등 시스템 자원이 매우 제한된 임베디드 시스템에서 이러한 자원을 효율적으로 사용하면서 적절한 성능을 제공하도록 자바 런타임 환경을 제공하는 것은 임베디드 시스템 분야의 매우 중요한 이슈이다. 자바 가상 머신을 구성하는 성분 중 가장 성능에 영향을 미치는 부분은 자바 바이트 인터프리터 부분이다. 현재까지 자바의 속도를 향상시키기 위해서 개발되어 온 자바 가속 기술은 소프트웨어 기술과 하드웨어 기술로 분류되며 이들은 모두 자바 인터프리터 부분의 성능 문제를 해결하는 데 초점을 두고 있다. 자바의 실행 속도를 향상시키기 위해 컴파일레이션에 기반한 소프트웨어 기술은 이식성이 저하되고 메모리 사용 증대로 인해 임베디드 시스템에 적용하는 데는 한계를 가지므로 최근에 전력 소모를 최소화하고 메모리 사용을 증가시키지 않는 마이크로프로세서를 확장한 하드웨어 기반의 자바 가속 기술이 많은 호응을 얻고 있다.

참 고 문 헌

[1] Steve Steele, Accelerating To Meet The Challenge of Embedded JAVA White Paper, ARM Limited Cambridge, Nov. 15, 2001.

[2] Boosting the Performance of Java Software on Smart Handheld Devices and Internet Appliances White Paper, Nazomi.

[3] Ipaul Manze, Insignia Solutions: Dynamic Adaptive Compilation and Competing Java Acceleration Strategies White Paper, July 30, 2001.

[4] Java™ Platform Micro Edition(J2ME™) Technology for Creating Mobile Devices White Paper, SUN, May 19, 2000.

[5] Ron Stein, Hardware Accelerators for J2ME Come of Age, *Java Developer's Journal*, Vol. 2, Issue 1, Jan. 2002, pp. 84 - 86.

[6] G. Chen, Tuning Garbage Collection in an Embedded Java Environment, *In proc. The USENIX Java Virtual Machine Research and Technology Symposium*, April 2001.