소형 기기들을 위한 자바가상머신의 성능향상을 위한 캐시의 설계 및 구현

백대현[0], 류헌수, 이정원, 이철훈
충남대학교 컴퓨터공학과
(dhbaek[0], hsryu, jwlee, chlee)@ce.cnu.ac.kr

A Design and Implementation of a Cache for the Performance Improvement of Java Virtual Machine for Small Devices

Dae-Hyun Baek[0], Hyun-Soo Ryu, Jung-Won Lee, and Cheol-Hoon Lee
Dept. of Computer Engineering, Chungnam National Univ.

요 약

최근들어 IT 산업이 급속도로 발전하면서, 리소스가 제한된 작은 기기들의 사용이 비약적으로 증가하는 추세에 있다. 이러한 임베디드 시스템이나 모바일 시스템과 같이 자원이 제한적인 기기들에 자바 환경을 적용하기 위해서 연구가 지속적으로 이루어지고 있다. 많은 기업들은 이러한 기기들에 실행되는 작은 크기의 Java™ Virtual Machine 구현하기 위해 CLDC(Connected, Limited Device Configuration) 표준에 기초하여 자바 가상 머신을 설계하고 구현한다. 작은 크기의 자바 가상머신은 크기 뿐만 아니라, 속도 또한 중요한 요소 중 하나이다. 본 논문에서는 Sun 의 CLDC 를 준수하는 자바 가상 머신의 속도 향상과 파워(Power) 절감 방법 중 하나인 캐시(Cache)를 설계하고 구현한 내용을 기술한다.

1. 서론

증가하는 임베디드 시스템에 대한 운영체제와 각각에 맞는 응용 프로그램들을 매번 다시 개발하는 것은 개발자에게는 매우 번거로운 일이다. 그러나 각각의 임베디드 시스템에 맞는 자바 가상 머신(Java Virtual Machine: JVM)이 탑재되어 있다면 하나의 응용 프로그램을 작성하더라도 모든 플랫폼에서 사용할 수 있게 된다[1]. 그러나 모바일 폰이나 PDA 들과 같은 기기들을 위해선 좀더 작은 용량의 자바 가상 머신이 필요하게 되었다. 1998 년부터 시작된 Spotless 프로젝트를 통해서 이러한 연구가 계속되었다[2]. 그들의 목적은 용량은 작게 구현하되 완벽한 자바 가상 머신의 기능을 수행하는 것이었다. 그 노력의 결과로써 Palm Pilot 을 타겟 플랫폼으로 하고 CLDC 표준의 기초가 된 KVM 이 만들어졌다[3]. KVM 이 탑재되는 소형 기기들은 자원이 제한적임에도 불구하고, 빠른 실행 속도와 적은 파워 소비가 중요 이슈가 되고 있다.

본 논문에서는 이러한 문제점을 조금이나마 해결할 수 있는 캐싱 기법을 기술하고, 구현한 내용을 기술한다. 여기에 사용된 캐싱 기법은 단순한 Deutsch-Schiffman Smalltalk-style 인라인 캐싱 기법이다.

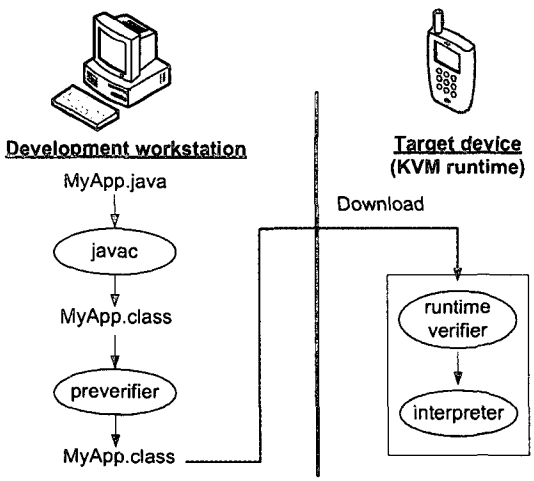
본 논문에서는 2 장에서 KVM 과 인라인 캐싱 정책에 대한 연구를, 3 장에서 인라인 캐싱의 설계 및 구현을, 4 장에서 테스트 환경 및 결과를, 5 장에서 결론 및 향후 과제를 기술한다.

· 본 논문은 한국과학재단이 지정한 지역협력연구센터(RRC)인 충남대학교 소프트웨어 연구센터의 지원으로 수행된 과제의 결과입니다.

2. 관련 연구

2.1 CLDC

CLDC 작고 리소스가 제한적이며 네트워크 연결능력이 있는 장치들을 위한 표준 자바 플랫폼을 정의하며, 자바 애플리케이션이나 내용들을 다른 장치들로 동적으로 전달하는 네트워크 모빌리티를 지원하도록 명세하고 있다.

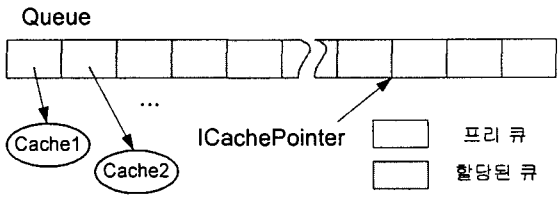


[그림 1] 두 단계 클래스 파일 검증과정

그리고 CLDC 에서는 장치들에게 부하를 줄일 수 있는 2 단계의 클래스 파일 검증을 한다. 이는 클래스 파일의 크기를 조금 늘리지만 런타임 실행 속도는 빨라진다. [그림 1]은 자바 애플리케이션을 컴파일하고, 그것이 타겟 플랫폼에서 실행되는 과정을 설명한다.

2.2 Deutsch-Schiffman Smalltalk style 캐싱

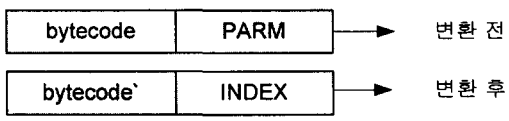
[그림 2]는 인라인 캐시 엔트리(Inline Cache Entry) 할당 과정을 설명한다. 첫 단계로 인라인 캐시 엔트리를 할당 하기 위해 ICachePointer 를 0 으로 초기화한다. 인터프리터가 특정한 바이트 코드를 실행하는 과정에서 캐시 엔트리의 생성을 요구하게 된다. 그 결과 인라인 캐시 엔트리가 생성되고, 그에 대한 포인터는 ICachePointer 가 가리키는 큐 위치에 저장된다. 다음 단계로 ICachePointer 값을 하나 증가시켜 다음에 인라인 캐시 포인터를 저장할 위치를 가르친다. 큐가 모두 할당되면 ICachePointer 를 0 으로 초기화하고 Cache1 엔트리를 프리시켜 다른 캐시 엔트리에 대한 포인터를 저장할 수 있게 한다.



[그림 2] 인라인 캐시 엔트리 할당 과정

3. 인라인 캐시(Inline Cache) 설계 및 구현

우선 캐시를 구현하기 위해서는 자바 가상 머신이 바이트 코드 변환을 지원해 주어야 한다. 자바 가상 머신이 처음으로 바이트 코드를 실행할 때, 특정한 바이트 코드는 그 보다 좀더 빠른 바이트 코드로 변환이 가능하다. 이 바이트 코드는 메모리 접근과 관련된 바이트 코드이다. 이렇게 변환된 바이트 코드는 다음에 다시 실행될 때 좀더 빠르게 실행된다. 이 중에서 캐시를 이용하는 바이트 코드는 극히 제한적이지만, 이는 자바 가상 머신이 실행될 때 많은 시간을 소비하는 함수와 클래스 검색(lookup)에 대한 시간을 줄여주는 기능을 수행한다. [그림 3]은 자바 가상 머신이 어떻게 바이트 코드를 변환 하는지에 대한 내용을 설명한다. PARM 에는 bytecode 를 위한 파라미터를 저장하고, INDEX 에는 인라인 캐시 인덱스 값을 저장한다.



[그림 3] 바이트 코드의 최적화 과정

3.1 캐시 구조(Cache Structure)

인라인 캐시 엔트리 구조는 [표 1]과 같이 정의할 수 있다. 각각의 인라인 캐시 엔트리에는 실행된 함수(method)와 클래스를 참조하는 포인터와 본래의 바이트 코드, 그에 상응하는 값이 저장되어 있다.

[표 1] ICACHE 구조

```
struct iCacheStruct {
    unsigned long* contents;
    unsigned char* codeLocation;
    short orignalParam;
    unsigned char originalInstrut;
};
```

-contents : 클래스와 인터페이스에서 정의한 함수나 클래스에 대한 포인터를 저장한다. 이는 함수나 클래스를 검색하는 과정에서 그 내용이 캐시에 저장되어 있다면, 복잡한 검색 과정을 거치지 않고 간단히 인라인 캐시 엔트리에 저장된 포인터 값을 리턴한다.

-codeLocation : 큐가 모두 할당되어 더 이상 인라인 캐시 에트리를 추가 하지 못할 때, 큐에 저장되어 있는 하나의 캐시 엔트리를 해제 해야만 한다. 이렇게 해제된 캐시 엔트리는 그에 상응하는 본래의 바이트 코드로 되돌려 주어야 하기 때문에 codeLocation 은 본래의 바이트 코드가 위치한 곳을 가르킨다.

-originalParam : [그림 3]에서 설명한 것과 같이 본래의 바이트 코드 다음에는 파라미터 값이 저장되어 있다. 그러나 바이트 코드를 변환하고 나면, 파라미터 값 위치에는 인라인 캐시 인덱스 값이 저장된다. originalParam 을 사용하는 목적은 본래의 바이트 코드로 되돌아 갈 때, 본래의 파라미터 값을 되돌려 주어야 하기 때문이다.

-originalInstrunt : 본래의 바이트 코드로 되돌려 주기 위해 바이트 코드를 저장하는데 사용한다.

3.2 인라인 캐시 엔트리 생성

[표 2] 인라인 캐시 엔트리 생성 알고리즘

```
int createICachEntry(*contents, *originalCode) {
    ICACHE iCache;
    if(Inline cache is full)
        releaseICacheEntry(ICachePointer);
    iCache = &InlineCache[ICachePointer];
    index = iCachePointer++;

    if(IcachePointer == CACHESIZE)
        iCachePointer = 0;

    iCache->contents = contents;
    iCache->codeLocation = originalCode;
    iCache->originalParam = getUnsignedShort
                            (originalCode+1);
    iCache->orignalInstrut = *originalCode;
    return index;
}
```

가상 머신이 함수나 클래스 룩업에 관련된 바이트 코드를 실행할 때, 가상 머신은 이에 대한 코드 최적화 과정과 함께 함수나 클래스에 대한 포인터를 저장하기 위해서 인라인 캐시 엔트리를 생성한다. [표 2]는 인라인 캐시 엔트리 생성 과정이다. 우선 큐에 빈 공간이 존재하는지 검사하여 빈 공간이 존재하지 않으면, ICachePointer 위치에 저장된 캐시 엔트리를 해제하고 새로 생성한 캐시 엔트리에 대한 포인터를 저장한다. 캐시 엔트리를 생성하는 과정에서 캐시 엔트리가 해제될 때를 대비하여 본래의 바이트 코드 값으로 되돌릴 수 있는 데이터들을 저장해야 한다.

3.3 인라인 캐시 엔트리 해제

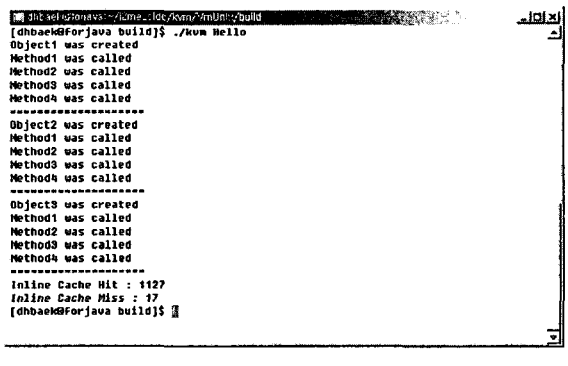
[표 3] 인라인 캐시 엔트리 해제 과정은 우선 주어진 ICachePointer 의 위치에 있는 인라인 캐시 엔트리에 해당하는 포인터를 큐에서 읽어온다. 다음 과정으로 그 데이터 구조에 저장되어 있는 본래의 바이트 코드의 위치와 값, 파라미터의 값을 본래의 값으로 되돌려주는 과정을 수행한다. 이 과정 이후 ICachePointer 위치에 다른 인라인 캐시 엔트리의 포인터를 저장할 수 있다.

[표 3] 인라인 캐시 엔트리 해제 알고리즘

```
releaseICacheEntry(index) {
    ICACHE iCache = &InlineCache[index];
    unsigned char codeL = (unsigned char*)
                          iCache->codeLocation;
    *codeL = iCache->originalInstrut;
    putShor(codeL+1, iCache->orignalParam;
}
```

4. 테스트 환경 및 결과

본 논문의 테스트 환경은 레드햇(Red Hat) 리눅스 9.0 운영체제 위에 자바 컴파일러로 J2sdk 1.4.2_05를 사용하였고, 자바 가상 머신으로 CLDC 1.0.4 를 사용하였다. 이러한 환경을 적용한 이유는 캐시와 관련된 부분은 시스템 독립적인 부분이기 때문이다.



[그림 4] Hello.java 실행 결과

자바 가상 머신을 컴파일 할 때 큐의 개수는 64개로 정의 했으며, 그 이외의 여러 가지 옵션들은 자원이 제한적인 기기에 적합하도록 설정하였다. 본 논문의 테스트에 사용된 프로그램은 Hello에 대한 객체를 3 개 생성하고, 각각의 객체는 Hello.java에 정의되어 있는 4 개의 함수를 순차적으로 호출하는 프로그램이다. [그림 4]는 Hello.java 프로그램을 실행한 결과 화면이다.

5. 결론 및 향후 과제

본 논문에서는 임베디드 시스템이나 모바일 시스템과 같이 자원이 제한적인 장치에 자바 가상 머신을 탑재할 때, 자바 가상 머신의 속도 향상과 파워 소비를 줄이기 위한 방법으로 함수나 클래스 룩업 과정을 보다 효율적으로 관리할 수 있는 인라인 캐싱 기법을 기술했다.

향후 연구과제로 현재 구현된 자바 가상 머신에는 Deutsch-Schiffman Smalltalk-style 인라인 캐싱 방법을 사용했다. 이 방법은 메모리의 소비는 작지만 캐시 히트율이 상대적으로 낮은 단점이 있다. 최근엔 메모리의 크기가 증가하고 있기 때문에 다소 메모리를 더 소비하더라도 캐시 히트율이 더 높은 캐시 기법을 적용하여 향상된 자바 가상 머신을 구현할 수 있게 하는 연구가 계속 되어야 한다.

6. 참고문헌

[1] http://www.inestech.com

[2] Nik Shayor, Douglas N. Simon, William R. Bush; *A Java Virtual Machine Architecture for Very Small Devices*, ACM SIGPLAN Conference on Language, compiler, and tool for embedded systems 2003, p34-41

[3] Sun Microsystems; *JSR-30 J2ME Connected, Limited Device Configuration*

[4] Tim Lindholm; *The Java™ Virtual Machine Specification Second Edition*, 1999

[5] Sun Microsystems; *KVM Porting Guide, CLDC, Version 1.0.4*

[6] Sun Microsystems; *Java™ 2 Platform Micro Edition Technology for Creating Mobile Devices*