자바 Ahead-of-Time 컴파일러와 해석기 간의 호출 오버헤드 평가

김익현°, 정동헌, 오형석, 문수묵
서울대학교 전기컴퓨터 공학부
ikhyun@altair.snu.ac.kr

Evaluation of Call Overheads Between Java Ahead-of-Time Compiler and Interpreter

Ik-Hyun Kim°, Dong-Heon Jung, Hyung-Sk Oh, Soo-Mook Moon
School of Electrical Engineering of Seoul National University

요 약

내장형 자바의 성능 향상을 위해 바이트코드를 기계어 코드로 미리 번역하여 내장형 시스템에 설치하는 Ahead-of-Time Compile(AOTC)가 많이 사용되고 있으나 수행 중에 동적으로 다운로드 되는 바이트코드를 수행하기 위해서는 기존의 해석기도 함께 사용되어야 한다. 이 경우 일부 자바 메소드는 AOTC에 의해 처리되고 일부 메소드는 해석기에 의해 수행되는 하이브리드 수행 환경이 된다. 이러한 환경에서 해석기 메소드가 AOTC 메소드를 호출하거나 AOTC 메소드가 해석기 메소드를 호출하는 경우 호출 오버헤드가 커서 성능을 저하시킬 수 가 있다. 본 연구에서는 AOTC에서 사용 가능한 두 가지 호출 인터페이스인 Java Native Interface(JNI)와 Compiled Native Interface(CNI)에 대해 하이브리드 수행 환경에서의 호출 오버헤드와 성능을 평가하고 각각의 장단점에 대해 논의한다.

1. 서 론

자바는 최근 디지털 TV, 핸드폰 등의 수많은 내장형 시스템에서 표준으로 채택되고 있다. 그 이유는 가상머신을 사용함으로써 다양한 플랫폼에서 일관된 실행 환경을 제공할 수 있고, 보안에 있어서도 안정적이며, 프로그램의 안정성을 높여주는 쓰레기 수집기능과 예외처리기술로 인해 개발이 쉽기 때문이다.[1]

그러나 자바는 바이트코드라는 중간 코드로 컴파일이 되기 때문에 하드웨어가 아니라 자바 가상머신(JVM)이라는 소프트웨어에 의해 해석기를 통하여 수행된다.[2] 이 소프트웨어는 하드웨어에 비해 매우 느리기 때문에 성능에 큰 문제점을 안고 있다.

이러한 문제를 해결하기 위해 바이트코드를 어셈블리 코드로 변환해서 수행하는, Just-In-Time Compiler (JITC)[3]나 Ahead-Of-Time Compiler (AOTC)[4-12]가 사용되고 있다. JITC에서는 바이트코드로부터 머신 코드로의 변환이 동적으로 일어나는 반면 AOTC에서는 정적으로 변환 작업이 일어나게 된다. 그러나 전력소비와 메모리 오버헤드 측면에서 많은 제약이 있는 내장형 시스템에서는 런타임 오버헤드가 큰 JITC보다 AOTC가 더 많은 이득을 가지고 있다. 국내 휴대폰에서 많이 채택하고 있는 WIPI 에서 자바를 수행하는 방식도 AOTC라고 볼 수 있다. (http://www.wipi.or.kr/)

AOTC는 두 가지의 구현방법이 있다. 첫 번째는 바이트코드를 머신 코드로 바로 변환시키는 방법(Java-to-native)[7-11]이고, 두 번째는 바이트코드를 C코드로 먼저 변환을 시킨 후 기존의 컴파일러를 이용해서 다시 머신 코드로 컴파일해주는 방법(Java-to-C 또는 bytecode-to-C)이다[4-6]. Java-to-native 방법은 어셈블리 단계에서 최적화 작업을 수행할 수 있지만, 이식성이 떨어지고, 개발하는데 많은 시간과 노력이 드는 반면에 Java-to-C 방법은 기존에 사용하던 C컴파일러를 이용해서 최적화 작업을 할 수 있기 때문에 더욱 실용적이고, 이식성이 뛰어나 많이 사용되고 있다.

그러나 동적으로 다운로드 받은 바이트코드 수행을 위해서는 AOTC와 함께 해석기도 사용되어야 한다. 이 경우 라이브러리 클래스들은 AOTC로 수행되고 동적으로 다운로드 받은 클래스들은 해석기로 수행된다. 이러한 환경에서 AOTC 메소드와 해석기 메소드 사이에 호출이 발생할 경우 호출 오버헤드로 인해 성능이 저하될 수 있다. 본 연구에서는 이 문제를 해결해보고자 AOTC에서 사용 가능한 두 가지 호출 인터페이스인 Java Native Interface(JNI)와 Compiled Native

Interface(CNI)에 대해서 하이브리드 수행 환경에서의 실험과 함께 두 가지 타입을 장단점을 비교 분석하였다. 이 논문의 구성은 다음과 같다. 2장에서는 Java-to-C AOTC의 동작 원리와 해석기 메소드와 AOTC 메소드 사이의 호출 오버헤드에 대한 간단한 설명을 한다. 3장에서는 JNI와 CNI의 기본 구조와 장단점에 대한 비교를 다룬다. 4장에서는 두 가지 호출 인터페이스 각각의 실험 결과를 분석하고, 5장에서 요약한다.

2. Java-to-C AOTC

2.1 동작 원리

Sun사의 CVM과 함께 돌아가는 Java-to-C AOTC 의 간단한 동작 구조는 그림 1과 같다. 먼저 AOTC가 바이트코드를 C코드로 변환하고, 그 C코드는 CVM 소스코드와 함께 GNU 컴파일러[13]로 컴파일된다. 여기서 우리의 AOTC는 코드 사이즈를 줄이기 위해 프로파일 피드백을 이용해서 클래스 파일에 있는 메소드들을 선택적으로 변환하게 된다. 이것은 AOTC가 해석기와 함께 동작하기 때문에 수행하는데 전혀 지장이 없다. 즉, AOTC 메소드와 해석기 메소드는 병렬적으로 수행된다. 이 방법은 클래스 파일을 동적으로 다운로드 받는 환경에서 유용하다. (예를 들면, 디지털 TV에서 케이블 라인을 통해 다운로드된 xlets가 해석기를 통해 수행되고, 자바 미들웨어는 AOTC로 수행된다.)

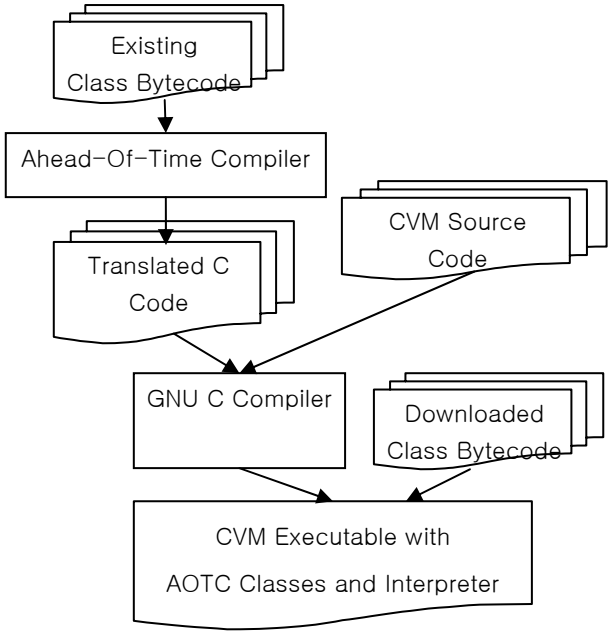

그림 1. Java-to-C AOTC의 동작 원리

Java-to-C AOTC의 구체적인 동작 순서는 다음과 같다. 먼저 바이트코드를 분석한 후에, 새롭게 선언해야 할 C 변수가 무엇인지 결정한다. 또 모든 로컬 변수와 스택 슬롯이 C 변수로 변환된다. 서로 다른 타입의 값들이 같은 스택 슬롯으로 푸쉬될 수 있기 때문에 로컬 변수의 타입이 스택 C 변수 이름에 같이 붙여진다. 예를 들면 s0_ref 는 레퍼런스 타입의 스택 슬롯 첫 번째 값을 뜻하고, s1_int 는 정수 타입의 스택 슬롯 두 번째 값을 뜻한다. 변수 설정이 끝나면 바이트코드들이 C 문장으로 한 줄씩 변환된다. 그 예로 바이트코드 iload_1의 변환과정을 살펴보자. 이 바이트코드는 정수 타입의 로컬 변수 1을 스택에 푸쉬하라는 의미이다. 만약 이때 스택 포인터가 첫 번째 슬롯을 가리키고 있다면, 이 바이트코드를 C 문장으로 변환한 결과는 s0_int = l1_int 와 같다. 이 과정에 따라 변환된 AOTC 메소드의 구체적인 예가 그림 2에 있다. 여기서 변환된 코드의 호출 인터페이스는 JNI이다.

(a) **자바 메소드**

```
public int max(int a, int b) {
   return (a>=b)? a:b;
}
```

(b) **바이트코드**

```
0: iload_0
1: iload_1
2: if_icmplt 9
5: iload_0
6: goto 10
9: iload_1
10:ireturn
```

(c) **변환된 C 코드**

```
Int Java_java_lang_Math_max_II(JNIEnv *env, int l0_int, int l1_int)
{
   int s0_int;
   int s1_int;
   s0_int = l0_int;                    // 0:
   s1_int = l1_int;                    // 1:
   if(s0_int < s1_int) goto L9         // 2:
   s0_int = l0_int;                    // 5:
   goto L10;                           // 6:
L9:    s0_int = l1_int;                // 9:
L10: return s0_int;                    // 10:
}
```

그림 2. 자바 코드와 변환된 C 코드의 예

2.2 해석기 메소드와 AOTC 메소드 사이의 호출 오버헤드

AOTC를 통해 생성된 코드는 시스템상에 정적으로 존재한다. 시스템에서 수행되는 모든 어플리케이션을 AOTC하게 되면 가상머신의 크기가 지나치게 커질 수 있다. 또한 자바에서는 동적으로 다운로드된 프로그램을 지원하게 되어있는데 이는 AOTC를 활용할 수 없다. 따라서 우리는 AOTC를 통해 미리 컴파일한 코드와 해석기를 함께 사용할 수 있는 환경을 고려해야 한다.

해석기와 AOTC가 공존하는 환경에서는 해석기 스택과 네이티브 스택이 번갈아 가며 사용될 수 있다. 이 두 환경에서 메소드 호출 시 인자를 전달하고 결과값을 받는 방법이 다르기 때문에 이를 처리하기 위한 과정이 필요하다. 이 과정에서 많은 호출 오버헤드가 발생할 수 있다.

우리는 이 문제를 해결하기 위해 호출 인터페이스 에 따른 성능을 비교해보았다. 다음 장에서 우리가 비교한 두 가지 호출 인터페이스, JNI와 CNI의 장단점을 분석한다.

3. 두 가지 호출 인터페이스의 비교 분석

3.1 JNI

3.1.1. JNI의 기본구조

JNI(Java Native Interface)란 자바로 만들어진 프로그램에서 해당 플랫폼에서만 실행 가능한 네이티브 코드에 접근하기 위해 만들어진 응용 프로그램 인터페이스이다[14]. 이는 자바 가상머신이 네이티브 메소드를 적재하고 수행할 수 있게 해주는데 보통 C와 C++에 대한 호출을 지원한다. 그림 3과 같이 JNI 호출 인터페이스로 AOTC된 메소드의 이름은 자바의 클래스명과 메소드 명의 결합으로, 인자는 가상머신과 정보를 주고받기 위한 환경변수 ee와 자바 메소드의 인자들로 하였다. 실제 연산에 사용되는 연산자 스택과 로컬 변수는 AOTC에 의해서 위치와 타입에 따라 별도의 이름을 가지는 변수로 변경되어 레지스터나 메모리에 영역을 갖는다. 마찬가지로 결과값의 전달도 레지스터나 네이티브 스택을 사용하는 방식으로 변경된다.

3.1.2 JNI의 장단점

일반적인 C언어로 작성된 코드처럼 레지스터나 네이티브 스택을 사용하므로 메소드가 호출 될 때 C 컴파일러에 의해 최적화되어 해석기에 비해 성능 향상을 기대할 수 있다. 또 자바 메소드 내에서 스택을 직접 접근하지 않기 때문에 보안적인 측면에서도 뛰어나다.

그러나 JNI 방식은 AOTC만이 존재하는 환경에는 최적의 성능을 기대할 수 있지만 동적인 클래스 로딩을 지원하기 위해 해석기가 추가된 환경에서는 해석기 메소드와 AOTC 메소드 사이의 호출 오버헤드로 인해 성능이 저하될 수 있다.

JNI에서는 메소드 내에서 해석기 스택에 직접 접근할 수 없게 설계되어있기 때문에 AOTC 메소드를 호출할 때에는 동적으로 해석기 스택에서 인자값을 복사해서 전달하는 과정이 필요하다. 해석기 스택에 어떤 타입의 인자가 몇 개가 있는지 루프를 돌면서 알아내고 이를 레지스터나 네이티브 스택에 차례로 넣어주게 된다. 그리고 메소드 수행이 끝나면 다시 레지스터나 네이티브 스택에서 결과값을 해석기 스택으로 복사해온다. 또 이 과정에서 레지스터 압박이 발생하기도 한다.

```
int java_lang_String__0003cinit(CVMExecEnv* ee, CVMStackVal32 *arguments)
```

```
Void java_lang_String__0003cinit (CVMExecEnv *ee, class cls_ICell, object l1_ref_ICell, int l2_int, int l3_int)
```

그림 3. 동일 메소드에서 CNI(위)과 JN(아래)의 호출 인터페이스

3.2 CNI

3.2.1 CNI의 기본구조

CVM에서 네이티브 메소드를 지원하기 위한 또 다른 호출 인터페이스로 CNI(Compiled Native Interface)가 있다. 기본적으로 JNI보다 가볍고 빠르게 동작하기 위해 만들어졌다. JNI와 비교할 때 가장 큰 차이점은 메소드 인자로 스택을 그대로 넘겨준다는 점이다. 그림 3처럼 가상머신과 정보를 주고받기 위한 환경 변수 뒤에 자바 메소드들의 인자로 스택의 포인터 만을 넘겨주고 있다. 그리고 호출된 메소드는 인자로 넘어온 스택의 포인터를 이용해 스택에 직접 접근한다.

3.2.2 CNI의 장단점

해석기에서는 인자 설정을 위한 별다른 작업 없이 바로 스택을 메소드의 인자로 넘겨줌으로써 JNI의 단점에서 언급한 많은 작업들을 생략할 수 있다. 대신에 AOTC 메소드 내에서 인자의 타입과 개수에 맞게 스택의 값을 로컬 변수에 할당해 주는 작업을 추가해야 하지만 이는 단순한 복사 명령으로 이루어져 있기 때문에 JNI의 인자 설정 작업에 비해 오버헤드가 작다.

그러나 JNI에 비해 CNI가 가지는 단점은 크게 두 가지가 있다. 먼저 AOTC 메소드에서 AOTC 메소드를

호출할 때 발생하는 오버헤드이다. 그림 4와 같이 기존에 JNI 에서는 AOTC 메소드에서 AOTC 메소드를 호출할 때 메소드 내에서 쓰는 변수들을 그대로 호출된 메소드의 인자들로 넘겨주면 되었지만, CNI에서는 인자로 전달될 변수들이 임시적으로 연산자 스택의 형태를 갖춘 후에 전달되기 때문에 오버헤드가 발생한다.

```
s2_ref = CNIjava_lang_StringBuffer_append (ee, s2_ref_lCell, s3_ref_lCell);
```

```
CVMStackVal32 callArg[2]; //임시 오퍼랜드 스택 생성
callArg[0] = s2_ref; //인자를 스택에 할당
callArg[1] = s3_ref;
java_lang_StringBuffer_append(ee, callArg);//함수호출
s2_ref= callArg[0];//결과값을 스택에서 읽음.
```

그림 4. AOTC 메소드에서 AOTC 메소드를 호출하는 부분(위는 JNI, 아래는 CNI)

두 번째 단점은 쓰레기 수집과 관련된 문제이다. AOTC에서는 레지스터 할당이 C 컴파일러에 의해 이루어지기 때문에 연산자 스택과 로컬 변수가 메모리상에 실제 저장될 위치를 알 수 없다. 따라서 쓰레기 수집이 일어날 가능성이 있는 곳에서의 쓰레기 수집 지도를 작성할 수 없게 된다. 이를 위해 메소드를 시작할 때 가상머신으로부터 오브젝트 포인터를 저장할 레퍼런스 스택을 할당 받아 런타임에 오브젝트 포인터 값을 생성된 C 코드 내의 변수와 함께 이 레퍼런스 스택에도 동시에 저장하는 방식을 택했다. CNI에서는 AOTC 메소드의 인자로 해석기 스택의 포인터만 넘어가기 때문에 메소드 내에서 해석기 스택의 값을 로컬 변수로 복사할 때, 레퍼런스 변수에 대해서는 추가적인 레퍼런스 스택을 할당 받아야 한다. 이 부분이 CNI의 두 번째 단점이 된다.

다음 장에서는 이번 장에서 살펴본 두 가지 인터페이스에 대한 비교 분석에 근거해서 하이브리드 환경에서의 실험 결과를 분석한다.

4. 실험 결과 및 분석

4.1 실험 환경

우리가 실험한 환경은 다음과 같다

운영체제 : Debian linux(Kernel 2.6.8-3)	
하드웨어 상세	
CPU model	Intel P4 2.4GHz(Single Processor)
Cache	512KB
Main Memory	512MB
Hard disk	40GB(7200RPM)

표 1. 실험 환경

벤치마크는 SPECjvm98(compress, jess, db, javac, mtrt, jack)을 사용하였고, 라이브러리 클래스들만 AOTC 해주는 Partial-AOTC와 라이브러리 클래스와 어플리케이션 클래스 모두를 AOTC해주는 Full-AOTC를 수행하였다.

4.2 실험 결과

SPECjvm98을 실험하기에 앞서서 두 가지 호출 인터페이스의 순수한 호출 오버헤드를 실험해보았다. 실험방법은 단순히 인자를 받아서 리턴만 해주는 테스트 메소드를 만들어서 그 메소드를 AOTC 한 다음, 해석기 메소드와 AOTC 메소드에서 인자의 개수를 달리하면서 테스트 메소드를 호출하는 방법을 이용하였다. 모든 실험은 7,000,000 번 호출해서 나온 결과이다.

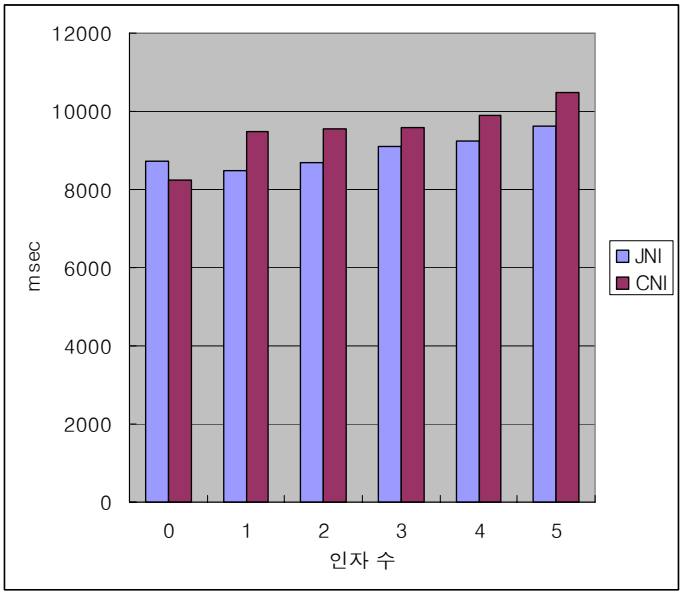

그림5. AOTC 메소드에서 AOTC 메소드로의 호출 오버헤드 측정 결과

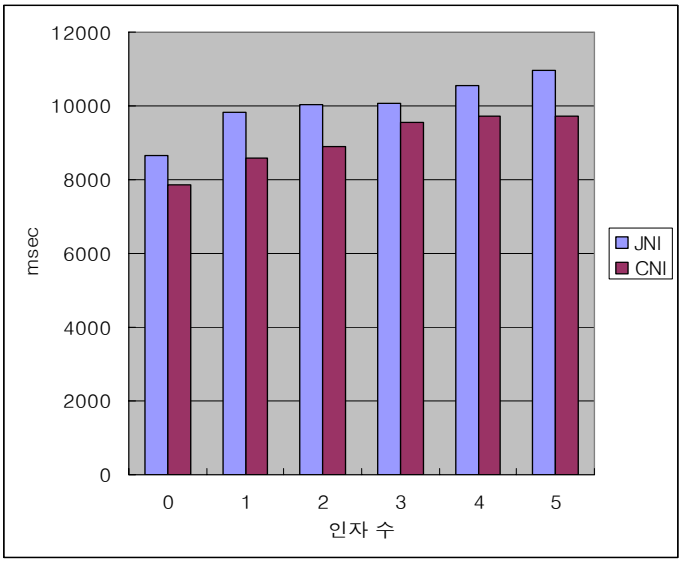

그림6. 해석기 메소드에서 AOTC 메소드로의 호출 오버헤드 측정 결과

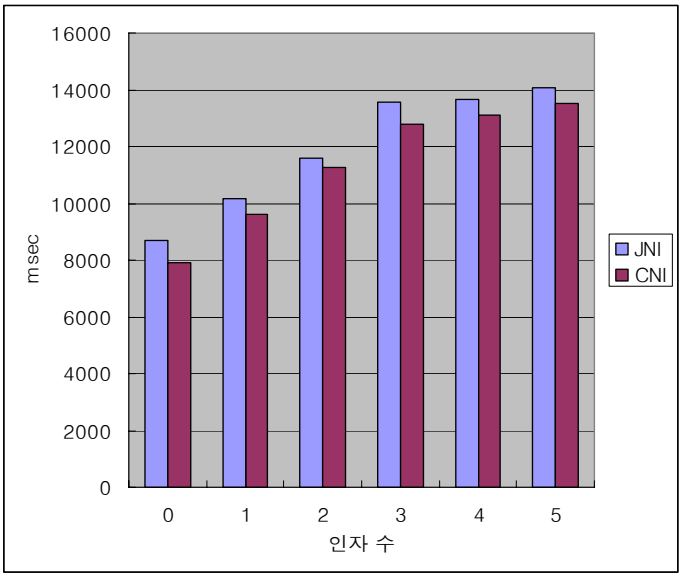

그림 7. 인자로 문자열 변수를 넘겨줬을 때 해석기 메소드에서 AOTC 메소드로의 호출 오버헤드 측정 결과

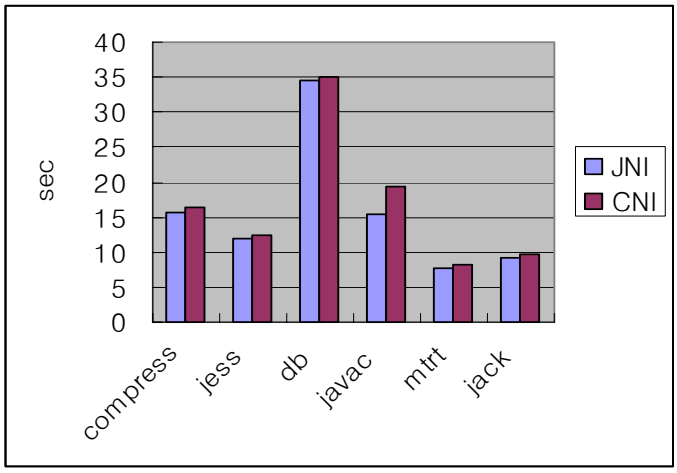

그림 8. Full-AOTC 수행 결과

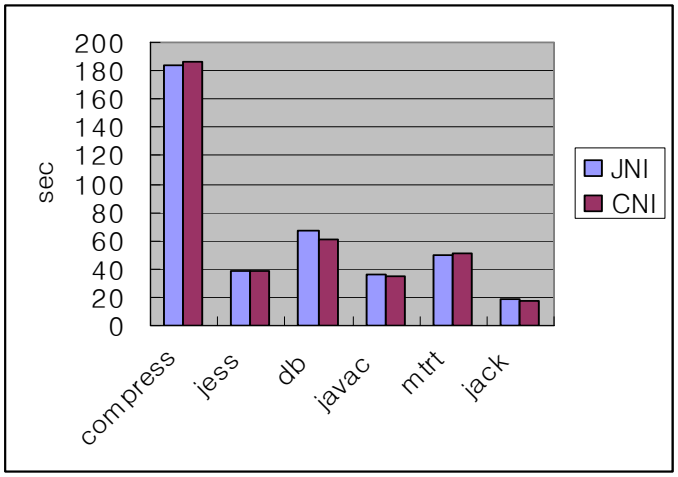

그림 9. Partial AOTC 수행 결과

그림 5, 6은 두 가지 인터페이스에 대해서 순수 호출 오버헤드의 측정 결과이다. 3장에서 분석했던 대로 AOTC 메소드에서 AOTC 메소드를 호출할 때 JNI의 성능이 더 좋고, 해석기 메소드에서 AOTC 메소드를 호출할 때 CNI의 성능이 더 좋다. 호출되는 메소드의 인자로는 모두 정수형 변수를 넘겨줬으며 레퍼런스 변수인 문자열 변수를 넘겨줬을 경우는 그림 7과 같이 CNI의 오버헤드가 늘어나 두 인터페이스의 성능 차이가 줄어들게 된다.

그림 8과 9는 실제 SPECjvm98에 대해서 Full AOTC로 수행했을 때와 Partial AOTC로 수행했을 때 두 호출 인터페이스의 성능 비교 그래프이다. Full AOTC에서는 모든 메소드 호출이 AOTC에서 AOTC로만 일어나기 때문에 JNI가 전체적으로 성능이 좋다. 그리고 Partial AOTC에서는 compress와 mtrt는 약간 느려졌으며, 나머지는 조금씩 빨라진 것을 알 수 있다. 그 이유는 다음 소개되는 표들을 보면 알 수 있다.

	해석기에서 AOTC로 호출	AOTC에서 AOTC로 호출
Compress	6,183	7,924
Jess	10,972,376	1,612,828
Db	65,966,855	512,831
Javac	24,068,504	3,665,398
Mtrt	5,534,858	843,391
Jack	17,657,107	7,590,268

표 2. Partial AOTC 내에서 메소드 호출 횟수.

	CNI	JNI
Compress	13,951	13,639
Jess	12,584,959	8,701,846
Db	66,479,530	66,463,864
Javac	27,733,726	25,790,971
Mtrt	6,169,434	979,763
Jack	25,247,219	24,135,724

표 3. AOTC 메소드 내에서 프레임 할당 횟수 비교

표 2는 Partial AOTC 내에서 각 벤치마크 별로 해석기에서 AOTC 메소드를 호출한 횟수와 AOTC 메소드에서 AOTC 메소드를 호출한 횟수를 측정한 값이다. 벤치마크들 중 compress만을 제외하고 해석기에서 AOTC 메소드로 호출하는 횟수가 더 많았고, db나 javac처럼 그 횟수가 많을수록 더 많은 성능이 향상되었다. 그러나 표 3에서 나타났듯이 호출된 AOTC 메소드 내에서 새롭게 할당되는 프레임이 많은 jess와 mtrt는 성능이 비슷하게 나오거나 오히려 더 나빠졌다.

4.3 . 추후 연구 방향

추후 연구는 두 호출 인터페이스의 장점을 모두 살릴 수 있는 방향으로 해야 한다. CNI이 JNI에 비해 해석기에서 AOTC 메소드로의 호출 오버헤드를 줄여줄 수 있으므로 이 인터페이스를 기본으로 하되 AOTC 메소드에서 AOTC 메소드를 호출할 때만 JNI 호출 인터페이스로 실행될 수 있도록 AOTC를 설계한다. 그리고 CNI에서 AOTC 메소드 내에서의 추가적인 레퍼런스 스택 할당이 발생하는 문제는 레퍼런스 변수를 인자로 넘겨줄 때 이미 레퍼런스 스택에 할당된 프레임 자체를 인자로 넘겨줄 수 있도록 해석기 스택 구조를 바꿔준다면 해결할 수 있을 것이다.

5. 결론

우리는 Java-to-C AOTC의 하이브리드 환경에서 성능향상을 위해 해석기 메소드와 AOTC 메소드 사이의 호출 오버헤드에 대한 연구를 하였다. AOTC에서 사용 가능한 두 가지 인터페이스인 JNI와 CNI의 장단점을 실험과 함께 비교 분석하였고 추후 연구 방향을 제시하였다.

6. 참고문헌

[1] Sun Microsystems, White Paper “CDC: An Application Framework for Personal Mobile Devices”

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

[3] J. Aycock. "A Brief History of Just-in-Time", ACM Computing Surveys, 35(2), Jun 2003

[4] T. A. Proebsting, G. Townsend, P. Bridges, J. H. Hartman, T. Newsham, and S. A. Watterson, "Toba: Java for Applications A Way Ahead of Time (WAT) Compiler," Proceedings of the Third USENIX Conference on Object-Oriented Technologies and Systems, Portland, Oregon, 1997

[5] G. Muller, B. Moura, F. Bellard and C. Consel, “Harissa: a Flexible and Efficient Java Environment Mixing Bytecode and Compiled Code”, In Proceedings of the 3rd Conference on Object-Oriented Technologies and Systems pages 1-20, Berkeley, June 16-20 1997.

[6] A. Varma, "A Retargetable Optimizing Java-to-C Compiler for Embedded Systems," MS thesis, University of Maryland, College Park, 2003

[7] M. Weiss, F. de Ferriere, B. Delsart, C. Fabre, F. Hirsch, E. Andrew Johnson, V. Joloboff, F. Roy, F. Siebert, and X. Spengler, “TurboJ, a Java Bytecode-to-Native Compiler”, 1998.

[8] M. Serrano, R. Bordawekar, S. Midkiff and M. Gupta, Quicksilver: A Quasi-Static Compiler for Java, In Proceedings of the ACM 2000 Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA’00). Pages 66-82, 2000.

[9] V. Mikheev, N. Lipsky, D. Gurchenkov, P. Pavlov, V. Sukharev, A. Markov, S. Kuksenko, S. Fedoseev, D. Leskov, A. Yeryomin, “Overview of Excelsior JET, a High Performance Alternative to Java Virtual Machines”, in Proceeding of the third international workshop on Software and performance WOSP’02, ACM Press. 2002.

[10] R. Fitzgerald, T. B. Knolock, E. Ruf, B. Steensgaard, and D. Tarditi, “Marmot: An Optimizing Compiler for Java”, Microsoft Technical Report 3. Microsoft Research, March 2000.

[11] Instantiations, Inc. JOVE: Super Optimizing Deployment Environment for Java, http://www.instantiations.com., July 1998

[13] Sun Microsystems, CDC CVM, http://java.sun.com / products/cdc

[13] GNU, Gnu Compiler Collection, http://gcc.gnu.org.

[14] Sun, Java Native Interface

http://java.sun.com/j2se/1.4.2/docs/guide/jni/