내장형 시스템을 위한 자바 AOTC의 설계와 구현

박종국[O] 정동헌 배성환 이재목 문수묵
서울대학교 전기컴퓨터공학부
{uhehe99[O], clamp, seasoul, jaemok, smoon}@altair.snu.ac.kr

Design and Implementation of Java AOTC for Embedded System

JongKuk Park[O], Dong-Heon Jung, SungHwan Bae, Jaemok Lee, Soo-Mook Moon
School of Electrical Engineering and Computer Sciences, Seoul National University

요 약

우리는 내장형 시스템에서 자바의 성능 문제를 해결하기 위해 수행시간 이전에(ahead-of-time) 자바 바이트코드를 기계어 코드로 변환하는 AOTC를 개발하였다.[1] 우리가 개발한 AOTC는 자바 클래스 파일을 읽어서 C코드로 변환한 후에 이를 C컴파일러로 컴파일하는 방식으로 동작한다. 이러한 방법으로 짧은 기간안에 하드웨어 아키텍처에 종속적이지 않으면서도 안정적인 고성능의 자바 실행 환경을 구축할 수 있었다. 본 논문에서는 AOTC의 전체 구조에서부터 C코드를 생성하는 방법 및 예외 처리와 쓰레기 수집기(GC)를 지원하기 위한 방법들에 대해서 다룬다. 또한, C컴파일러에 의한 최적화의 한계를 극복하기 위해 자바에 특화된 최적화를 AOTC에 포함하였는데 이에 대해서도 설명한다. 우리의 AOTC를 Sun Microsystems의 J2ME CDC VM인 CVM위에 적용해본 결과 벤치마크에 따라서 평균 5~14배의 성능 향상을 관찰 할 수 있었다.

1. 서 론

자바는 95년 처음 도입된 이후 빠른 속도로 사용이 확산되고 있으며 최근에는 인터넷 뿐 아니라 디지털TV나 휴대폰등의 내장형 플랫폼에서도 표준으로 채택되고 있다.이는 프로그램이 아키텍처 독립적인 바이트코드(bytecode)형태로 저장된 후 자바 가상머신(자바VM)위에서 수행되는 자바 고유의 성질에서 얻어지는 이식성과 보안성에 힘입은바가 크다.

이처럼 바이트코드를 사용하면 뛰어난 이식성과 보안성을 얻을 수 있지만 한편으로는 인터프리트하는 과정에서 발생하는 성능 저하 문제가 생겨난다. 이를 해결하기 위해서 프로그램 수행시간에 자바VM에 의해 바이트코드가 기계어 코드로 번역되는 Just-In-Time(JIT)컴파일 기법과 수행 시간 이전에 미리 바이트코드를 기계어 코드로 변환해 놓는 방식인 Ahead-Of-Time컴파일(AOTC)기법이 연구 개발 되었다.

이 중 JIT컴파일 기법은 자바언어의 동적인 특성을 가장 잘 반영해주는 기법으로 현재 널리 사용되고 있지만 컴파일이 프로그램 수행시간에 이루어지기 때문에 저전력/저성능/작은메모리를 기반으로 하는 내장형 시스템에 적용하기에는 무리가 있다.

이러한 이유로 내장형 시스템에서는 AOTC가 주로 사용되고 있다. AOTC는 자바VM이 수행되기 전에 미리 자바 클래스파일을 기계어 코드로 변환하는 기법으로 이를 통해 변환된 자바 메소드들은 인터프리터를 거치지 않고 직접 CPU위에서 수행된다.

2. C코드를 생성하는 방식의 AOTC의 장점

AOTC를 사용해서 자바 바이트코드를 기계어 코드로 바꾸는 방법에는 직접 기계어 코드를 생성하는 방식[1-4]과 C코드를 생성한 후 이를 GCC와 같은 일반 C컴파일러를 사용해서 컴파일하는 방식[5-8]의 2가지가 있다. 우리는 이 중 C코드를 생성하는 방식을 선택했는데 이 방식은 기계어 코드를 직접 생성하는 방식에 비해서 다음과 같은 장점이 있다.

- C코드를 생성하는 방식은 기계어 코드를 생성하는 방식과 비교했을 때 상위 레벨의 언어만을 다루므로 더 짧은 시간에 안정적인 시스템을 구축할 수 있다.
- 기존 C컴파일러의 전통적인 최적화 기법들을 모두 활용할 수 있기 때문에 모든 최적화들을 일일이 구현할 필요가 없다.
- C코드를 생성하게 되면 기계어 코드를 직접 생성하는 방식에 비해서 다른 아키텍처를 사용하는 플랫폼으로의 이식이 간단하다
- C언어용으로 개발된 각종 디버거와 프로파일러들을 모두 활용할 수 있다. 또한 호스트 아키텍처가 타겟 아키텍처와 다를 경우에도 호스트 머신에서 대부분의 디버깅이 가능하다.

이와 같은 이유로 우리는 차후 내장형 시스템을 위한 자바 가속기술로 C코드를 생성하는 방식의 AOTC가 주류가 되리라고 생각한다. 본 논문에서는 C코드를 생성하는 방식의 AOTC를 설계할 때 고려해야 할 다음과 같은 문제들에 대해서 다룰 것이다.

[1] 본 연구는 삼성 전자의 연구비 지원을 받아 수행되었다.

첫 째는 성능 향상을 위한 최적화이다. 우리의 AOTC에서는 C컴파일러로 컴파일을 하는 단계에서 C컴파일러의 최적화가 적용되기 때문에 모든 종류의 최적화를 구현할 필요가 없다. 하지만, C컴파일러의 최적화는 C언어로 작성된 프로그램에 적합하게 설계되었으므로 AOTC에서 생성한 C코드를 최적화하는데 한계가 존재한다. 우리는 메소드 인라이닝을 비롯한 자바 언어에 특화된 최적화들을 구현했으며 이를 통한 성능 향상을 살펴보았다.

둘 째는 자바 언어의 특성 때문에 생겨나는 문제들이다. 자바는 시스템의 보안과 사용자의 편리를 위해 예외(exception) 처리와 쓰레기 수집기(GC)를 지원하므로 자바VM에서도 이를 필수적으로 지원해야 한다. 우리는 이를 기존의 방식과는 다른 효율적인 방식으로 처리하였으며 이에 대해서도 서술할 것이다.

3. AOTC의 개요

3.1. AOTC의 전체 구조

그림 1은 우리가 개발한 AOTC가 탑재된 자바VM의 전체 구조를 나타낸다. (a)는 클래스파일을 C코드로 바꾸는 컴파일러를 (b)는 이를 목적 코드로 컴파일하는 GCC를 (c)는 AOTC와 함께 수행할수 있도록 수정된 Sun Microsystems의 J2ME CDC의 자바VM인 CVM을 나타낸다.[9]

AOTC를 포함하는 자바VM은 기계어로 변환된 메소드 뿐 아니라 네트워크를 통해서 전송되는 클래스 파일역시 수행할 수 있어야 한다. 이를 위해서는 인터프리터와 컴파일된 메소드간의 상호 호출 및 예외 처리가 가능해야 하는데 이를 지원하기 위해서 CVM의 인터프리터를 일부 수정하였다.

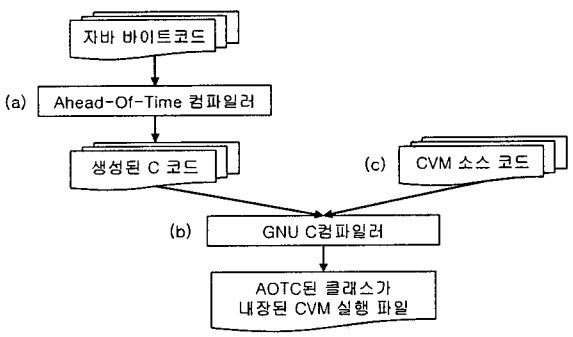

그림 1. AOTC의 전체 구조

3.2. 코드 생성

바이트 코드를 C코드로 변환하는 과정은 다음과 같은 순서로 이루어진다. (1)바이트코드를 읽어서 CFG(control flow graph)를 작성한다. 이 때 바이트코드는 컴파일 과정에서 필요한 정보를 포함할 수 있는 IR(intermediate representation)로 변환된다. (2)자바 스택을 시뮬레이션해서 자바 오퍼랜드 스택의 값을 C언어의 변수로 변환한다. (3)각종 최적화를 수행한다. (4)CFG를 순회하면서 IR을 C코드로 변환한다.

그림 2는 (a)최대값을 리턴하는 간단한 자바 소스코드와 (b)이에 대한 바이트 코드, (c)이를 AOTC해서 생성한 C코드, (d)이 C코드를 GCC컴파일러로 컴파일해서 최종적으로 생성된 기계어 코드를 보여준다. AOTC된 C코드에는 바이트코드 0, 1을 번역하면서 자바 오퍼랜드 스택변수에 값을 로드하기 위해 s0_int, s1_int에 값이 복사되지만 이를 GCC를 사용해서 컴파일 한 후에는 이와 같이 불필요한 복사명령이 제거된 최적화된 코드가 생성됨을 확인할 수 있다.

(a) 자바 코드

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

(b) 바이트코드

```
0: iload_1
1: iload_2
2: if_icmplt  7
5: iload_1
6: ireturn
7: iload_2
8: ireturn
```

(c) AOTC된 C 파일

```
JNIEXPORT CVMJavaInt JNICALL Java_Test_max(CVMExecEnv*
ee, jclass cls_1Cell, CVMJavaInt l1_int, CVMJavaInt
l2_int)
{
    CVMJavaInt s0_int; // stack변수를 선언
    CVMJavaInt s1_int; // stack변수를 선언

    s0_int = l1_int;                    // 0: iload_1
    s1_int = l2_int;                    // 1: iload_2
    if (s0_int < s1_int) {              // 2: if_icmplt 7
        goto LL_Java_Test_max_7;
    }
    s0_int = l1_int;                    // 5: iload_1
    goto EXIT;                          // 6: ireturn

    LL_Java_Test_max_7:
    s0_int = l2_int;                    // 7: iload_2
    goto EXIT;                          // 8: ireturn

    EXIT:
    return s0_int;
}
```

(d) GCC를 거쳐서 생성된 목적 코드

```
0:      slt     v0,a2,a3
4:      lw      v1,0(a1)
8:      bnezl   v0,34 <Java_Test_max+0x10>
c:      move    a2,a3
10:     jr      ra
14:     move    v0,a2    // delay slot
```

그림 2. 자바 코드와 생성된 C코드 및 기계어 코드

4. 성능향상을 위한 최적화

GCC의 최적화 기법들은 C언어로 작성된 프로그램에 적합하도록 설계되었기 때문에 자바에서 유래한 C코드를 최적화하는데 한계가 존재한다. 따라서 우리는 다음과 같이 GCC가 수행할 수 없는 자바에 특화된 최적화만을 선택적으로 구현하였다.

- **정적 프로파일링**
  JIT컴파일러와 달리 AOTC에서는 수행시간 이전에 미리 코드를 생성하기 때문에 동적 프로파일링을 사용할 수 없다. 따라서 우리는 정적 프로파일링을 사용하였다. 이를 통해 각 메소드가 수행되는 위치와 횟수를 얻어서 그 결과를 메소드 인라이닝과 컴파일할 메소드를 선택하는데 사용하였다.
- **중복된 검사 코드의 제거**
  널 포인터 검사, 배열 범위 검사, 클래스 초기화 검사는 시스템의 안정성을 보장해 주지만 이미 검사된 항목에 대해서도 중복해서 검사가 일어나기 때문에 시스템의 성능을 저하시킨다. 따

라서 데이터플로우분석(data flow analysis)을 사용해서 이미 검사된 항목에 대한 중복된 검사를 제거해주었다.

- **루프 벗겨내기(loop peeling)**
  AOTC과정 중 루프안에있는 중복된 검사 코드를 제거하지 못하는 경우가 종종 생겨나는데 이는 자바에서 예외의 유무, 위치, 순서 및 종류가 정확(precise exception semantic)해야 하기 때문이다. 따라서, 루프안에 있는 중복된 코드를 제거하기 위해서 루프를 한겹 벗겨내는 방식[10]을 적용하였다.
- **메소드 인라이닝**
  객체 지향 언어에서는 작은 크기의 메소드가 빈번하게 수행되기 때문에 메소드 인라이닝이 최적화에 미치는 영향이 크다. 따라서 GCC보다 더 적극적인 인라이닝을 수행해야 한다. 우리는 정적 메소드와 동적 메소드를 모두 인라이닝하였고 이를 통해 최대 30%, 평균10% 가량의 성능 향상을 얻을 수 있었다.
- **GNU C 확장 기능의 사용**
  우리의 AOTC는 GCC의 성능에 많은 영향을 받으므로[6] GCC컴파일러의 성능을 극대화 하기 위해 GNU C의 확장기능(extension)을[11] 일부 사용하였다. 예를 들어, 널 포인터 예외가 자주 발생하지 않는다는 사실에 착안해 GCC가 코드 배치(reordering)를 효율적으로 할 수 있도록 분기 예측(branch prediction) 정보를 제공하였다. 이는 표준C만으로는 불가능한 일이다.

5. 예외 처리와 쓰레기 수집기의 지원

5.1. 예외 처리

C코드를 생성하는 방식의 기존 AOTC는 모두 setjmp/longjmp를 사용해서 예외 처리를 구현하였다.[5, 6, 8] 하지만 이러한 방식에서는 실제로 예외가 발생하지 않을 때에도 계속해서 setjmp를 수행해야 하는 문제점때문에 좋은 성능을 얻기 힘들었다. 우리는 setjmp/longjmp를 사용하지 않는 방식으로 예외를 처리했으며 이를 통해 1.3~154%가량의 성능 향상을 얻을 수 있었다.

5.2. precise 쓰레기 수집기(GC)의 지원

C코드를 생성하는 방식의 기존 AOTC는 모두 conservative GC[5, 6, 8]만을 지원하였는데 최근에는 응답 속도가 빠르고 100% 메모리 회수가 가능한 precise GC가 더 선호 되고 있다.[9] 우리는 메소드 안에서 사용되고 있는 객체의 참조를 최대한으로 줄여서 최소한의 오버헤드로 precise GC를 지원하였다.

6. 실험 결과

6.1. 성능 향상

우리가 개발한 시스템을 인텔 Pentium4 2.4G, 512MB RAM, HaansoftLinux 2005의 환경에서 테스트 해 보았다. 벤치마크로는 CaffeineMark와 SpecJVM98이 사용되었으며 GCC 3.4.3에서 최적화옵션 O2를 사용하였다.

그림 3은 AOTC를 적용했을 때의 성능 향상을 나타낸다. Loop, Logic, compress처럼 최적화의 여지가 많은 벤치마크에서의 성능 향상이 큰 반면, String, db처럼 인터프리터에서 수행되는 코드의 비율이 적은 벤치마크에서는 성능 향상은 작은 편이었다.

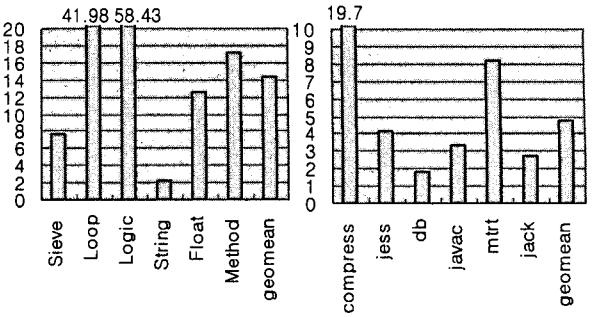

그림 3. CaffeineMark와 SpecJVM98에서의 성능 향상 배수

6.2. 코드 사이즈 증가

표 1은 AOTC후의 코드 사이즈 증가를 나타낸다. 실험 결과는 벤치마크 메소드와 이를 수행하는데 필요한 시스템 클래스의 메소드 중 1번 이상 수행된 것들을 모두 AOTC했을 때 얻어진 것이다.

표 1. AOTC후의 코드 사이즈 증가

	인터프리터 실행파일	벤치마크 클래스파일	AOTC된 코드가 포함된 인터프리터
CaffeineMark	2097KB	9KB	2372KB
SpecJVM98	2097KB	2364KB	6316KB

7. 결론

JIT컴파일 기법을 적용하기 힘든 내장형 시스템 환경에서는 AOTC가 자바 가속을 위한 방법으로 자주 사용된다. 우리는 C코드를 생성하는 방식의 AOTC를 개발해서 높은 생산성과 이식성을 갖춘 시스템을 구축하였다. 이를 Sun의 J2ME CVM에 적용해 본 결과 평균 5~14배의 성능 향상을 얻을 수 있었다.

8. 참고문헌

1. Jove: super optimizing deployment environment for Java, http://www.instantiations.com/_NewSite/jove/JOVEReport.pdf
2. Robert Fitzgerald, et al., *Marmot: An Optimizing Compiler for Java,* Software Practice and Experience, 30(3) p, 199-232, 2000
3. Mauricio Serrano, et al., *Quicksilver: a quasi-static compiler for Java,* ACM SIGPLAN Notices, 35(10) p, 66-82, 2000
4. V. Mikheev, et al., *Overview of excelsior JET, a high performance alternative to java virtual machines*, in *Proceedings of the 3rd international workshop on Software and performance*, 2002
5. Todd A. Proebsting, et al., *Toba: Java for Applications A Way Ahead of Time (WAT) Compiler*, in *Proceedings of the Third USENIX Conference on Object-Oriented Technologies and Systems*, 1997, Portland, Oregon
6. Gilles Muller and Ulrik Pagh Schultz, *Harissa: A Hybrid Approach to Java Execution,* IEEE Software, 16(2) p, 44-51, 1999
7. Michael Weiss, et al., *TurboJ, a Java Bytecode-to-Native Compiler*, in *Proceedings of the Workshop on Languages, Compilers, and Tools for Embedded Systems*, 1998
8. Ankush Varma and Shuvra S. Bhattacharyya, *Java-through-C Compilation: An Enabling Technology for Java in Embedded Systems* in *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition Designers (DATE)*, 2004
9. CDC Foundation Porting Guide, http://java.sun.com
10. Daniel J. Scales, et al., *The Swift Java Compiler: Design and Implementation,* WRL Research Report 2000/2, 2000
11. GCC online documentation http://gcc.gnu.org/onlinedocs/