제32회 추계학술발표회 논문집 Vol. 32, No. 2(I)

자바 Ahead-of-Time 컴파일러에서의 효율적인 예외처리 방법

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

Efficient Exception Handling in Java Ahead-of-Time Compilation

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

요 약

자바는 이식성과 보안의 장점으로 인하여 내장형 시스템에서 널리 사용되고 있으나 인터프리터를 통한 바이트코드의 수행으로 인하여 성능이 저하되는 문제를 포함하고 있다. 이를 해결하기 위한 한 방법으로 수행시간 전에 바이트코드를 기계어 코드로 미리 번역하여 수행시간에는 기계어 코드가 수행되도록 하는 Ahead-of-Time 컴파일러 (AOTC)가 사용되고 있다. 특히 바이트코드를 C 코드로 변환한 다음 기존의 컴파일러를 이용하여 기계어 코드를 생성하는 방식을 많이 택하고 있다. 본 논문에서는 AOTC에서 효율적인 예외처리 (exception handling) 기법을 제안한다. 기존의 AOTC 에서는 예외를 발생하는 메쏘드와 예외를 처리하는 메쏘드가 다른 경우 setjmp/longjmp를 이용하여 예외처리를 수행하고 있으나 우리는 메쏘드 호출 후의 예외 검사를 통해 예외처리를 수행한다. 우리는 제안된 예외처리 방법은 Sun의 CDC 가상 머신을 위해 개발된 AOTC 에 구현되었으며 SPECjvm98 벤치마크에서의 실험을 통해 setjmp/longjmp 방식에 비해 1.3%에서 154%까지의 성능향상이 가능함이 확인되었다.

1. 서론

현재 자바는 이식성과 보안의 장점 그리고 프로그램 개발의 용이성으로 인해 휴대폰, 디지털 TV, 홈 네트워크, 텔레매틱스 등 다양한 내장형 시스템의 소프트웨어 플랫폼으로 사용되고 있다. 특히 이식성의 장점은 자바 프로그램이 자바가 플랫폼 독립적인 바이트 코드로 컴파일 되어 배포되기 때문인 데, 문제는 바이트코드는 CPU에서 직접 수행되지 않고 자바 가상 머신 상에서 인터프리트 되기 때문에 느릴 수 밖에 없다는 점이다. 이러한 성능 문제를 해결하기 위한 방법의 하나로 수행시간 전에 바이트코드를 기계어 코드로 미리 번역하여 수행시간에는 기계어 코드가 수행되도록 하는 Ahead-of-Time 컴파일러 (AOTC)이 사용되고 있다. 특히 바이트 코드를 C 코드로 번역하고 기존의 최적화 C 컴파일러를 이용하여 기계어 코드를 생성하는 방식이 많이 사용되고 있다 [1][2][3].

AOTC를 설계하는 데 있어서 중요한 문제 중의 하나의 자바의 예외처리에 대한 번역이다. 자바는 try block 과 catch block 을 제공하여 try block 에서 예외가 발생하면 catch block 에서 그 예외를 catch 하여 처리하도록 되어 있다. 그런데 메쏘드 A에 있는 try block 내에서 메쏘드 B를 호출하고 호출된 메쏘드 B에서 예외가 발생하거나 혹은 메쏘드 B에서 호출한 메쏘드 C에서 예외가 발생하는 경우에도 메쏘드 A의 catch block에서 예외를 처리해야 한다. 즉 예외가 발생하면 현재 수행 중인 메쏘드에 해당하는 catch block 이 없다면 호출 스택 (call stack) 을 따라가며 그 예외를 처리할 수 있는 catch block을 가진 메쏘드를 찾는 작업이 필요하다. 기존의 AOTC에서는 이를 처리하기 위해 catch block 이 있는 메쏘드에는 setjmp() 를 수행하고 예외가 발생한 지점에서는 longjmp()가 수행되도록 번역하고 있다. 그러나 우리가 관찰한 바로는 setjmp와 longjmp의 수행 빈도가 높아서 이러한 구현은 효율적이지 못하다고 판단되었다. 그래서 우리는 메쏘드 호출에서 귀환할 때 마다 예외가 발생 했는 지를 검사하도록 번역하는 방식을 제안한다. 본 논문에서는 이러한 방식을 setjmp/longjmp 방식과 비교 평가한다.

이 논문은 다음과 같이 구성되었다. 2장에서는 이전의 Bytecode-To-C AOTC에서 사용하던 setjmp/longjmp를 이용한 예외처리에 대해서 설명하고, 3장에선 효율적인 예외처리 방법을 제안하고 설명하겠다. 4장에선 실험결과를 제시하고 그 결과를 분석하겠다.

2. 기존의 Bytecode-to-C AOTC의 예외 처리 방법

기존의 Bytecode-to-C AOTC[1][2][3]에선 setjmp/longjmp를 이용한 Stack Cutting방법[4]을 이용하였다. C 에서 메쏘드 영역을 넘어서는 제어 흐름을 변경할 수 있는 명령어는 setjmp/longjmp뿐이기 때문이다.

stack cutting[4]방법은 catch block이 있는 메쏘드에서 setjmp()를 수행한다. setjmp를 수행하게 되면 현재 프로세스의 정보를 포함한 jmpbuf가 생성된다. 생성된 jmpbuf는 예외가 발생한 경우에 longjmp를 수행하기 위해서 jmpbuf의 리스트를 관리 해야 한다. 기존의 예외 처리 방법은 setjmp를 수행하는 위치에 따라 두 가지로 나뉜다. 첫 번째는 메쏘드의 프롤로그에서 수행하는 것이고, 두 번째는 try block의 입구에서 수행하는 것이다.

그리고 예외가 발생한 경우엔 발생된 오브젝트 값을 글로벌 변수에 저장하고 jmpbuf를 이용해서 longjmp를 수행한다. longjmp는 jmpbuf에 저장된 프로세스의 정보를 모두 복원한다. 스택, 레지스터 정보, PC를 복원하기 때문에 마치 setjmp를 수행한 지점으로 점프하는 것과 같은 효과를 가진다.

setjmp/longjmp를 이용한 예외 처리 방법은 아래의 비효율성을 가진다.

- setjmp 수행: SPEC JVM98 벤치마크를 AOTC한 경우 setjmp는 매우 자주 수행되며 그 오버헤드도 크다. setjmp는 우리의 예외 처리 방법에서 사용되는 단순한 if 검사에 비해 약 250배 정도의 수행시간이 걸린다.
- lock과 jmpbuf의 리스트: longjmp를 수행한 후에 예외를 처리하는 메쏘드와 예외가 발생한 메쏘드 사이의 모든 메쏘드의 lock을 제거해야 한다. 그러기 위해서 try block이 없는 메쏘드인 경우에도 synchronized 메쏘드는 항상 lock을 프롤로그에서 추가하고 에필로그에서 제거해야 한다. 그리고 longjmp를 수행하기 위해 jmpbuf를 유지해야 한다.
- 자바 지역 변수: longjmp를 수행하면 레지스터의 모든 값들이 복구 된다. 그렇기 때문에 자바 지역 변수는 레지스터에 사용 할 수 없고 값을 항상 메모리에 저장해야 한다.
- catch block 찾기: setjmp를 메쏘드의 프롤로그에서 수행하는 경우에 catch block을 찾는 오버헤드가 존재한다. 예외가 발생하여 longjmp로 돌아온 후에 발생된 예외가 어떤 try block내에서 발생한 것인지 확인해야 한다. 그러기 위해선 bytecode program counter(bpc)가 필요하다. try block의 입구에서 setjmp를 수행하는 경우엔 try block에 속한 catch block으로 점프하면 된다.

우리는 이전의 bytecode-to-C AOTC에서 사용된 setjmp/longjmp를 이용한 예외 처리 방법보다 좀 더 효율적이고 간단한 방법을 제안하겠다.

3. 우리의 예외 처리 방법

우리의 예외 처리 방법은 두 가지 경우로 나누어진다.

예외가 발생하는 메쏘드 내에서 예외 처리가 가능한 경우엔 catch block으로 점프한다. catch block에서는 예외를 처리할 수 있는지 확인한 후에 처리할 수 있으면 catch block의 코드를 수행한다. 또한 AOTC를 수행하는 동안에 발생 가능한 예외와 catch block이 처리할 수 있는 예외를 알 수 있다. 그러므로 catch block이 발생 가능한 예외를 처리할 수 있다고 확신할 수 있을 경우엔 catch block에서 확인을 할 필요 없이 바로 catch block의 코드를 수행하도록 한다.

만약 예외를 같은 메쏘드 내에서 처리를 할 수 없을 경우엔 메쏘드 에필로그로 점프한 후에 발생된 예외의 오브젝트를 글로벌 변수에 저장하고 현재 메쏘드를 종료한다. Caller 메쏘드에서 Callee 메쏘드에서의 귀환 후에 확인한다. 예외가 발생이 확인되면 적절한 catch block으로 점프한다. 메쏘드의 귀환 후엔 항상 예외를 확인 해야 하는 오버헤드가 존재한다. 하지만 메쏘드 호출 후의 예외 검사는 단순한 if 명령어를 이용한 확인으로 성능을 크게 떨어뜨리지 않는다.

그리고 우리의 AOTC에선 메쏘드 호출 후 수행되는 검사의 오버헤드를 더 줄일 수 있다. 메쏘드 호출 후에는 예외의 발생 유무를 확인하는 것뿐 아니라 GC의 발생도 확인 하여야 한다. 우리의 AOTC를 구현한 CDC 자바 가상 머신은 Precise Generational GC를 사용한다. 그리고 semispace copying GC 알고리즘과 mark-compact GC 알고리즘을 사용하기 때문에 오브젝트들이 옮겨 질 수 있다. 그러므로 GC가 발생한 후에는 현재 메쏘드 뿐만 아니라 Caller 메쏘드의 레퍼런스의 값들을 항상 복구시켜주어야 한다. 그러기 위해서 함수 호출 직후에 GC발생을 확인하여야 한다. 우리는 GC 확인과 예외 확인을 하나의 확인으로 합쳐 예외 확인으로 인해 발생되는 오버헤드를 줄일 수 있다.

4. 실험

4.1. 실험 환경

우리는 Bytecode를 C 코드로 변역하고 기존의 최적화 C 컴파일러를 이용하여 기계어 코드를 생성하는 방식의 Bytecode-to-C AOTC[1]를 CDC 자바

[1]본 연구는 삼성 전자의 연구비 지원을 받아 수행되었다.

가상 머신에 개발하였다. C 컴파일러는 이미 널리 사용되는 GNU C 컴파일러를 이용하였다. 실험은 Linux 2.6.9-1hs가 설치된 2400Mhz/512MB의 인텔 펜티엄 4 프로세서에서 수행하였다.

벤치마크는 SPEC JVM98을 사용하였다. 아래의 테이블은 SPEC JVM98의 예외 발생 횟수와 예외 발생 지점으로부터 예외를 처리하는 지점까지의 Distance이다.

Bench	Exception Count	Distance
compress	7	2
jess	10	2
db	10	2
javac	22408	1.854338
mtrt	7	2
jack	241934	2.513863

표 1. SPEC JVM98의 특성

표1의 Compress, Jess, DB 와 Mtrt에서 발생되는 예외는 UnsupportedEncodingException으로 벤치마크가 아닌 OS로 인해 의해 발생하는 예외들이다. 위의 표1에서 보듯이 javac, jack에서 많은 예외가 발생하며 jack은 예외 발생 지점으로부터 예외를 처리하는 지점까지의 Distance가 크다.

4.2. 실험 결과

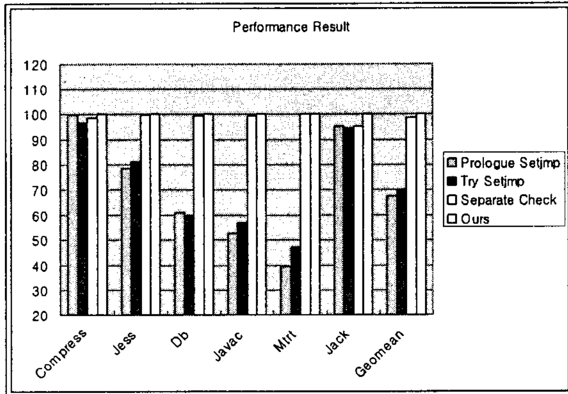

그림 1. SPEC JVM98 실험 결과

위의 그림은 각 예외 처리 방법을 적용하여 SPEC JVM98을 AOTC한 경우의 성능이다. 우리의 예외 처리 방법이 SPEC jvm98의 모든 벤치마크에서 좋은 성능을 보여준다. 우리의 예외 처리 방법이 다른 방법에 비해 1.25%에서 154.08% 정도의 성능 향상을 보인다. 위의 Separate Check는 GC와 예외 체크를 분리 시킨 코드의 성능이다. Separate Check와 Ours의 성능 차에서 볼 수 있듯이 메쏘드 호출 후의 예외 확인의 오버헤드가 상당히 적다.

Compress의 성능은 큰 차이가 없다는 것을 알 수 있다. 그 이유는 Compress는 예외가 발생하지 않고 또한 try block이 거의 없기 때문에 일반 패스에서 setjmp를 수행하는 횟수가 적기 때문이다.

Bench	Prologue setjmp	Try block setjmp
compress	4,082	4,780
jess	20,957	36,903
db	159,670	159,878
javac	1,405,076	2,059,950
mtrt	109,923	33,782
jack	3,771,659	4,352,106
Geomean	141,294	142,972

표 2. 프로그램 실행 중에 수행되는 setjmp 횟수

하지만 mtrt의 경우 횟수가 매우 적지만 성능은 많이 떨어진다. 그 이유는 mtrt는 다른 벤치마크에 비해 수행 시간이 짧다. 그러므로 setjmp의 오버헤드가 크게 미치게 된다.

Jack은 setjmp의 횟수가 크지만 비교적 성능이 높다. 그 이유는 예외가 자주 발생하고 또한 예외가 발생하는 메쏘드와 핸들하는 메쏘드 사이의 Distance가 크기 때문이다. 우리의 방법은 메쏘드의 예외가 발생하게 되면 GC가 발생하지 않았더라도 모든 레퍼런스 값을 복구해야 한다. 그러므로 오버헤드가 커지게 되어 Jack과의 성능 차이가 적어진다.

5. 결론

기존의 Bytecode-to-C AOTC의 예외처리는 setjmp/longjmp를 이용하였다. 하지만 setjmp/longjmp를 이용한 예외처리 방법은 효율적이지 않기 때문에 우리는 간단하고 효율적인 예외 처리 방법을 제안하였으며 이를 통해 1.3에서 154%의 성능 향상을 얻을 수 있었다.

6. 참고 문헌

[1]Todd A. Proebsting, Gregg Townsend, Patrick Brides, John H. Hartman, Tim Newsham and Scott A. Watterson, Toba: Java For Applications A Way Ahead of Time (WAT) Compiler

[2]JOVE: Super Optimizing Deployment Environment for Java, 1995.

[3]Ankush Varma, "A Retargetable Optimizing Java-to-C Compiler for Embedded Systems," master of science thesis, 2003

[4]Takeshi Ogasawara, Hideaki Komatsu and Tosho Nakatani, A Study of Exception Handling and Its Dynamic Optimization in Java, OOPSLA 2001