□ 기술해설 □

자바 가상 머신을 위한 JIT 컴파일 기법

서울대학교 양병선*·박성배·문수묵**

1. 개 요

프로그래밍 언어 자바는 썬 마이크로시스템즈에서 개발한 언어로서, 객체지향성, 묵시적 메모리 관리와 쓰레기 처리 지원, 다중 쓰레드의 지원, 예외 처리의 지원, 높은 이식성 등을 장점으로 인터넷을 통해서 광범위하게 사용되고 있다.

어떠한 프로세서 혹은 하드웨어에서도 동작할 수 있는 높은 이식성은 자바의 큰 장점인데, 이것은 자바로 작성된 프로그램의 경우 생성되는 목적코드가 특정한 프로세서에 맞춰져 있지 않고 자바 가상 머신이라는 가상의 명령어 집합과 실행환경에 맞춰져 있기 때문이다. 자바 가상 머신의 명령어 집합을 바이트코드라고 부른다. 바이트코드의 크기를 줄이기 위해서 자바 가상 머신은 모든 연산에 있어서 스택을 이용한다. 스택을 이용하게 되면, 일반적인 마이크로프로세서 명령어에서 레지스터를 나타내는 부분이 없어지게 되므로 각 명령어의 크기가 줄어들게 된다.

모든 자바 프로그램은 자바 가상 머신에 의해서 실행되며 따라서 자바 프로그램을 실행하는 성능을 향상시키기 위해서는 자바 가상 머신을 효율적으로 구현해야 한다. 자바 가상 머신의 구현에는 여러가지 기법이 있으며 이 글에서는 그러한 기법들 중에서 특별히 JIT 컴파일러 기법을 소개하고자 한다.

2. 자바 가상 머신의 구현 방식

자바 가상 머신을 구현하는 방식은 바이트코드를 실행하는 방식에 따라 크게 둘로 나눌 수 있는데, 바이트코드를 직접 실행하기 위해 설계된 자바 칩을 이용하는 것과, 자바 칩을 이용하지 않고 일반적인 마이크로프로세서를 이용해서 소프트웨어로 구현하는 방식이 있다. 소프트웨어로 구현하는 경우 또 다시 둘로 나눌 수 있는데, 하나는 바이트코드 하나 하나를 읽고 해석해서 실행하는 방식이 있고, 나머지 하나는 실행해야 할 바이트코드로부터 마이크로프로세서의 고유 코드를 생성해 낸 뒤 이 코드를 실행하는 방식이 있다. 이때 전자를 해석기(interpreter) 방식이라고 부르고, 후자를 JIT(Just-In-Time) 컴파일러 방식이라고 부른다.

자바 칩을 이용하게 되면 빠른 응답 시간 등의 장점은 있지만, 자바 칩이 아닌 보통의 마이크로프로세서들의 더 강력한 성능을 이용하지 못하는 단점이 있다. 현재 대부분의 자바 가상 머신은 자바 칩을 사용하지 않고 일반적인 범용 마이크로프로세서에서 소프트웨어를 이용한 방식으로 구현되어 있다.

해석기 방식으로 바이트코드를 실행하게 되면, 프로그램의 전체적인 응답 시간이 상대적으로 짧아지고 프로그램의 수행에 끊김이 없는 장점이 있지만, 각각의 바이트코드를 읽고 해석해서 적절한 작업을 실행하는 과정에서 전체적인 수행시간이 매우 길어지게 된다. 반면 JIT 컴파일러 방식을 이용하게 되면, 상대적으로 빠른 고유 코드를 직접 실행하므로 각 메소드의 실행시간은 짧아진다. 그러나 바이트코드를 고유 코드로 컴파일하는 과정에서 많은 시

* 학생회원
** 종신회원

간이 소요되어서, 프로그램의 응답 시간이 길어지고, 프로그램의 수행이 JIT 컴파일을 하는 동안 끊기게 되는 등의 단점 때문에 컴파일된 메소드가 충분히 많이 사용되지 않으면 해석기 방식보다 더 느린 경우도 발생한다.

3. JIT 컴파일러의 소개

바이트코드의 실행을 위해서 JIT 컴파일 기법을 사용하는 경우, 먼저 실행하려는 바이트코드를 읽어 들인 뒤 이 바이트코드에서 고유 코드를 생성해 낸 다음, 생성된 코드를 실행하게 된다. 따라서 전체 실행 시간은 바이트코드로부터 고유 코드를 생성해 내는 시간과 생성된 코드를 실행하는 시간 두 가지의 합에 의해 결정된다. 그러나 한번 생성된 코드는 다음번 해당하는 바이트코드가 실행될 때 다시 사용될 수 있으므로 고유 코드를 생성하는데 필요한 시간이 전체 프로그램의 실행에서 차지하는 비중은 상대적으로 작아질 수 있다.

그림 1은 현재 사용되고 있는 JIT 컴파일 방식에 대한 설명이다.

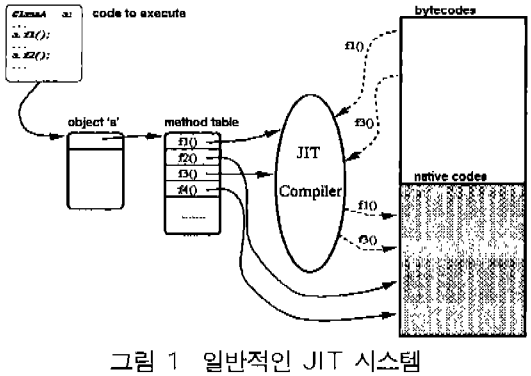


그림 1 일반적인 JIT 시스템

JIT 컴파일러는 메소드 단위로 코드 생성을 수행한다. 그림 1에서 보이는 것처럼 ClassA의 메소드 테이블의 각 엔트리는 초기에 JIT 컴파일러 모듈을 가리키고 있다. 프로그램 도중에 메소드가 호출되면 메소드 테이블에 따라서 JIT 컴파일러가 호출되고, 호출된 JIT 컴파일러는 해당하는 메소드에 대한 고유 코드를 생성해 낸 뒤, 메소드 테이블의 값을 생성된 고유 코드를 가리키도록 바꿔주고 메소드를 실행한다. 그러면 메소드가 다시 호출되었을 때 메소드 테이블은 생성된 고유 코드를 가리키고 있으므로 자연스럽게 기존에 생성된 고유 코드가 실행이 되게 된다.

해석기 방식에 비해서 JIT 컴파일 방식이 빠를 수 있는 이유는 생성된 고유 코드가 여러번 다시 호출될 수 있기 때문인데, 만일 메소드들이 여러번 호출되지 않는다면, JIT 컴파일러 기법이 해석기보다 오히려 더 느릴 수 있다. 또, 모든 메소드를 실행하는데 있어서 JIT 컴파일을 하는 시간이 필요하므로, 메소드가 호출된 시점부터 메소드의 첫 코드가 실행되는 시간까지 차이가 생기게 된다.

다음은 JIT 컴파일의 예제이다. 좌측은 바이트코드이며 우측은 그에 해당하는 고유 코드이다.

표 1 JIT 컴파일의 예제

바이트 코드	고유 코드	
iload-0 iload-1 iadd	ld [$fp-0], $r0	#iload-0
	st $r0, [$sp+0]	#iload-0
	ld [$fp-4], $r1	#iload-1
	st $r1, [$sp + 4]	#iload-1
	ld [$sp+0], $r0	#iadd
	ld [$sp+4], $r1	#iadd
	add $r0, $r1, $r2	#iadd
	st $r2, [$sp+0]	#iadd

4. JIT 컴파일러의 여러 가지 기법

4.1 코드 생성 방식에 따른 분류

JIT 컴파일러가 바이트코드를 고유 코드로 변환하는 방법은 여러 가지가 있다. 특히 자바의 스택을 고유 코드에서 어떻게 접근하게 할 것인가가 중요한 문제이다.

Kaffe 등의 자바 가상 머신이 채택하고 있는 방식은 자바 스택을 메모리에 위치시키는 것이다. 즉, 자바 스택이 그대로 메모리의 특정한 영역에 1대 1로 대응되도록 한 뒤, 바이트코드가 자바 스택의 일정 영역에서 읽거나 쓰면, 그에 해당하는 고유 코드의 메모리에서 읽거나 쓰게 만드는 것이다. 이 경우 대부분의 모든 연산이 메모리를 통하게 되므로 레지스터를 이

용하는 고유코드에 비해서 성능이 떨어지게 된다. 그러나 바이트 코드를 복잡한 분석없이 즉시 코드를 생성할 수 있으므로 상대적으로 컴파일에 걸리는 시간이 짧다.

자바 스택을 메모리에 위치시키는 방법의 단점을 개선한 방법이 자바 스택을 레지스터에 1대 1로 대응시켜서 실행하는 방법이다. 이러한 접근이 가능한 배경에는 자바의 메소드들이 사용하는 최대의 스택 크기가 상대적으로 작다는데 있다. 따라서 자바 스택 전체를 레지스터에 대응시키는 것이 가능하며, 따라서 자바 스택을 메모리에 대응시키는 방법에 비해 컴파일 시간의 차이는 거의 없으면서 코드의 질을 향상시킬 수 있는 방법이다. 이 경우 앞의 방법에 비해 메모리 연산이 대폭 줄어들게 되므로 코드의 질이 쉽게 향상될 것임을 기대할 수 있다.

첫째와 둘째 방법은 바이트코드가 그대로 하나 하나의 명령어로 바뀌게 된다. 그러나 자바 코드에서 많은 부분을 차지하는 지역 변수에 대한 접근 명령은, 만일 레지스터 할당을 바이트코드로부터 생성된 고유 코드에 대해 적용한다면 없어질 수 있는 명령들이 대부분이다. 셋째 방법은 바이트코드로부터 중간 코드를 생성해 내고 이 중간 코드에 대해 레지스터 할당 및 기타 최적화 기법을 적용하는 방법이다. 이 방법은 기존의 컴파일러에서 사용하는 방법과 거의 차이가 없으나 레지스터를 할당할 때 코드의 질 보다는 컴파일 속도가 더 중요해지므로 새로운 빠른 레지스터 할당 알고리즘을 필요로 하며, 기존의 최적화 기법들을 컴파일 시간에 따라 선택적으로 사용해서 컴파일 시간을 줄이는 차이가 있다.

4.2 JIT 컴파일 적용 방식에 따른 분류

앞에서 지적한 것처럼 JIT 컴파일 방식은 프로그램의 응답 시간을 증가시키는 단점이 있다. 이 단점을 보완하기 위해서 JIT 컴파일을 언제 어떠한 메소드에 대해서 적용할 것인가에 따라 몇가지 기법이 제안되거나 구현되어 있다.

기본적인 JIT 컴파일러는 실행하려는 모든 자바의 메소드에 대해 고유코드를 생성한 뒤 이 생성된 코드를 실행한다. 실행되는 모든 메소드는 한번의 코드 생성 과정을 거치게 되고, 따라서 코드 생성에 걸리는 시간을 보상할 수 있을 만큼 자주 메소드가 실행되지 않으면 해석기 방식에 비해 오히려 실행에 걸리는 시간이 증가할 수 있다.

Smart JIT 컴파일 기법은 해석기 기법과 JIT 컴파일 기법을 함께 사용하는 방법이다. 이 기법은 실행회수가 적은 메소드들에 대해서는 해석기 기법을, 실행회수가 많은 메소드에 대해서는 JIT 컴파일 기법을 사용하는 방법이다. 가령 클래스 초기화 메소드들은 단 한번 실행되고 다시 실행되지 않으므로, 이러한 메소드들은 해석기로 실행을 하는 것이 결과적으로 이득이 된다.

연속적 컴파일 기법에서는 자바 가상 머신이 두 개의 쓰레드를 생성해서 하나의 쓰레드는 바이트코드를 실행하고, 하나의 쓰레드는 JIT 컴파일을 하도록 한다. 바이트코드를 실행하는 쓰레드는, 실행하려는 메소드가 이미 JIT 컴파일이 되어 있다면 생성된 코드를 실행하고 그렇지 않다면 해석기를 통해 메소드를 실행한다. 그동안 컴파일하는 쓰레드는 실행되는 자바 클래스에 속해있는 모든 메소드들을 컴파일한다. 이 방법은 실행하는 쓰레드가 각종 입출력을 기다리는 시간동안 컴파일 하는 쓰레드가 동작할 수 있으므로 결과적으로 JIT 컴파일의 시간 지연이 감추어질 수 있게 되어서, 다른 JIT 컴파일 기법에 비해 상대적으로 전체 프로그램의 응답 시간이 짧아지는 효과가 있다.

그러나 연속적 컴파일 기법은 소속된 클래스의 모든 메소드를 컴파일하므로 실행되지 않는 메소드들도 컴파일하게 될 가능성이 높고, 따라서 전체적인 시스템의 속도를 저하시킬 수 있다. 향상된 JIT 컴파일 기법은 smart JIT 컴파일 기법과 연속적 컴파일 기법의 장점을 취합하는 기법으로, 연속적 컴파일의 두 개의 쓰레드와 smart JIT의 선택적 JIT 컴파일을 합친 것이다. 향상된 JIT 컴파일 기법은 연속적 컴파일 기법의 단점인 필요없는 메소드에 대한 컴파일 시간을 줄일 수 있으며, smart JIT에서 컴파일을 하는 시간이 연속적 컴파일 기법과 같이 감추어 질 수 있으므로 전체적인 응답 속

도와 수행 속도의 향상을 기대할 수 있다.

5. 남은 일

전통적인 컴파일러 최적화 기법들 중에서 JIT 컴파일러에서 사용되기에 좋은, 즉 짧은 시간 동안에 큰 코드 성능 향상을 얻을 수 있는 기법들을 찾아내어 이를 JIT에 적용하는 작업이 필요하며, 특히 많은 시간이 걸리는 레지스터 할당을 빠르게 수행할 수 있는 새로운 알고리즘이 필요하다.

자바 가상 머신의 성능을 향상시키기 위해서 JIT 컴파일 기법이 매우 중요한 위치를 차지하는 것은 사실이지만, 쓰레드와 쓰레드간의 동기화와 쓰레기 처리(garbage collection)도 자바 프로그램의 실행시간 중에서 많은 부분을 차지한다. 따라서 효율적인 자바 가상 머신의 구현이라는 궁극적 목표를 위해서는 JIT 컴파일 기법 뿐 아니라 위에 다른 부분에 대한 최적화가 필수적이다.

참고문헌

[1] F. Yellin and T. Lindholm, *The Java Virtual Machine Specification,* Addison-Wesley, 1996.

[2] G. Gosling, B. Joy and G. Steele, *The Java Language Specification,* Addison-Wesley, 1997.

[3] C. A. Hsieh, M. T. Conte, T. L. Johnson, J. C. Gyllenhaal and W. W. Hwu, "Using NET to Capture Performance in Java-Based Software," 1997.

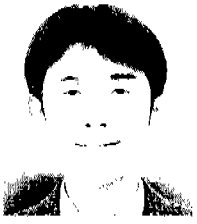
[4] M. P. Plezbert and R. K. Cytron, "Does Just in Time = Better Late than Never?," *Principles of Programming Languages,* 1997.

[5] M. P. Plezbert, "Continuous Compilation for Software Development and Mobile Computing," Washington University, 1996.

[6] P. R. Wilson, "Uniprocessor Garbage Collection Techniques," *International Workshop on Memory Management,* 1992.

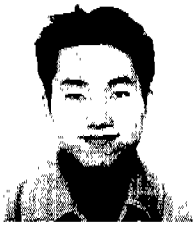
[7] F. Yellin and T. Lindholm, "Java Runtime Internals," JavaOne Sun's Worldwide Java Developer Conference, 1997.

[8] 양병선, 문수묵, "향상된 JIT 컴파일 기법을 이용한 Java 가상 머신의 설계" 한국정보과학회 제24회 추계발표대회, 1997.



문 수 묵

1987 서울대학교 컴퓨터공학과(학사)
1990 University of Maryland 전산학과(석사)
1991～1993 IBM T. J. Watson 연구소 객원연구원
1990 University of Maryland 전산학과(박사)
1993～1994 Hewlett-Packard Company 소프트웨어 엔지니어
1994～현재 서울대학교 전기공학부 조교수
1997 IBM T. J. Watson 연구소 방문연구원
관심분야 : 최적화 컴파일러, 마이크로프로세서 구조



박 성 배

1997～현재 서울대학교 전기공학부(학사), 서울대학교 전기공학부 석사과정
관심분야 : 최적화 컴파일러, 마이크로프로세서 구조, dynamic compilation

양 병 선

1997～현재 서울대학교 전기공학부(학사), 서울대학교 전기공학부 석사과정
관심분야 : 최적화 컴파일러, 마이크로프로세서 구조, dynamic compilation