디스크 캐쉬 기법을 이용한 자바 가상 기계의 성능 향상

강정욱○ 김철기 이준원
한국과학기술원 전산학전공

Performance Enhancement of Java Virtual Machine using Disk Cache Mechanism

Jeong-uk Kang Cheol-gi kim Joon-won Lee
Division of Computer Science
Korea Adavanced Institute of Science and Technology

요 약

자바는 이동성, 호환성, 안정성을 위해서 자바 가상 기계를 이용한 수행 방법을 사용한다. 이는 수행 속도를 저하 시키는 한 요인이다. 그래서, 정적 컴파일 모델과 동적 컴파일 모델과 같은 자바 가상 기계의 성능을 높이기 위한 여러 연구들이 진행되었다. 정적 컴파일 모델은 자바의 특성을 해치며, 동적 컴파일 모델은 초기화 시간이 증가하는 문제점이 있다. 본 논문에서는 디스크 캐쉬 기법을 이용하여 동적 컴파일 모델에서 발생하는 초기화 시간을 줄이는 자바 가상 기계를 구현하였다. SPEC JVM98[1]을 이용하여 Kaffe[2] 와 비교했을 때, 초기화 시간이 2배에서 5배 정도 줄었음을 알 수 있었다.

1. 서론

자바는 이기종간의 분산 컴퓨팅과 내장형 시스템을 위하여 설계된 언어로 시스템 비종속적 특성과 메모리 안정성을 보장하는 특성을 가지고 있다. 이러한 특성은 대중화된 웹 분야뿐 아니라 분산 응용 프로그램 및 이식성 있는 응용 프로그램 개발에 적합하다[3].

자바는 자바 바이트코드(bytecode)를 이용하여 자바 가상 기계가 해석(interpret)하는 방식으로 수행한다. 때문에 자바를 기존의 C나 C++같이 네이티브 코드 (native code or machine code)로 컴파일 하여 수행하는 프로그램과 비교하여 느린 성능을 보이는 원인이 된다. 자바의 성능 개선 방법은 하드웨어(hardware)적인 접근 방법과 소프트웨어적인 접근 방법으로 분류할 수 있다. 하드웨어적인 접근 방법은 성능을 가장 좋게 향상시킬 수 있으나 추가적인 하드웨어 비용이 든다. 소프트웨어적인 접근방법으로는 대표적으로 정적 컴파일 모델(static compilation model)과 동적 컴파일 모델(dynamic compilation model)로 다시 나뉜다. 정적 컴파일 모델은 좋은 성능을 제공하지만, 자바의 표준을 지키지 못 한다. 동적 컴파일 모델은 자바의 표준을 지키면서 성능을 향상시킬 수 있는 장점이 있지만, 초기화 시간이 증가하는 문제점이 있다.

본 논문에서는 정적 컴파일 모델의 호환성 문제를 해결하면서 동적 컴파일 모델의 초기화 시간을 단축하여 반응 시간을 개선하는 혼합 컴파일 모델의 자바 플랫폼을 제안하고 구현한다. 또한, 네이티브 검증기를 통하여 사용자에게 바이트코드를 직접 수행하는 것과 같은 투명성을 제공하며,네이티브 코드의 안전성 보장 기법을 제공해서 악의적인 공격으로부터 저장된 네이티브 코드를 보호한다.

본 논문의 구성은 다음과 같다. 2.절에서는 관련 연구로서 자바 가상 기계의 성능 향상을 위한 연구들을 서술하고 3.절에서는 본 논문에서 구현한 디스크 캐쉬의 구조와 디스크 캐쉬에 저장된 네이티브 코드의 버전 관리 시스템의 구조를 설명하고, 이를 이용하여 자바 가상 기계의 성능 향상을 위한 기법을 제시한다. 4.절에서는 자바 가상 기계의 성능을 비교 분석하고, 5.절에서는 결론과 향후 연구 과제를 기술한다.

2. 관련 연구

정적 컴파일 모델은 자바 소스 코드나 자바 바이트코드를 WAT(Way-Ahcad-of-Tim) [4]와 같은 컴파일러를 통해서 사용자가 실행 파일을 미리 만든다. 컴파일 시간이 수행 시간에 포함되지 않으므로 공격적인 최적화 기법들을 많이 사용할 수 있다. 이 모델은 성능을 향상시키기 위해서 자바의 호환성과 시스템 비종속성을 포기한 접근이다. 때문에 성능은 C나 C++과 비슷하지만 자바의 표준을 만족시키지 못한다. 이러한 특성으로 인해서 성능은 좋지만 실제 자바 가상 기계에 적용된 적은 없다. 비슷한 연구로는 Caffcinc[5]과 Harissa[6]가 있다.

동적 컴파일 모델은 대표적으로 JIT(Just-In-Time) 컴파일[7]과 JIT 컴파일의 일종인 HotSpot[8] 기술이 있다. JIT 컴파일은 메소드가 호출되었을 때 메소드의 자바 바이트코드를 네이티브 코드로 컴파일한 후 수행하는 방법을 말한다. 정적 컴파일 모델과는 다르게 네이티브 코드를 저장하지 않으므로 매번 응용 프로그램이 수행될 때 마다 같은 컴파일 작업을 반복하여 초기화 시간이 길어지는 문제가 있다. 이는 수행 시간을 증가시킬 뿐 아니라 사용자와 상호작용에서의 반응 속도를 저하시킨다. 또한, 수행 시간에 컴파일 시간이 포함되어 컴파일 시간에 커다란 제한이 있기 때문에 C나 C++과 같은 언어에서 사용되는 여러 가지 최적화 기법의 적용이 힘든 단점이 존재한다. 이를 극복하기 위한 방법으로 HotSpot 기술이 나오게 되었다. 처음에는

인터프리터 방식으로 바이트코드를 수행하면서, 프로파일링(profiling) 기법을 통해 수행 중의 상황을 감시하고 있다가, 자주 수행되는 부분을 공격적인 최적화 기법을 적용하여 네이티브 코드로 컴파일 하여 자바 가상 기계의 성능을 향상 시킨다. 하지만 매번 수행할 때 마다 프로파일링을 위한 초기 수행 시간이 필요하므로 서버 어플리케이션과 같이 긴 수행 시간을 가지는 응용 프로그램에 적합한 방법이다. 이와 비슷한 연구로 Annotation-Aware Just-In-Time 컴파일러[9]와 NET(Native Executable Translation)[10] 컴파일러가 있다.

3. 디스크 캐쉬 기법을 이용한 자바 가상 기계

3.1 제안하는 디스크 캐쉬 기법을 이용한 자바 가상 기계 구조

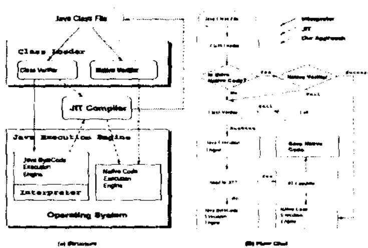

그림 1: 제안하는 디스크 캐쉬 기법을 이용한 자바 가상 기계 구조

그림 1은 본 논문에서 구현한 디스크 캐쉬 기법을 이용한 자바 가상 기계의 구조이다. 본 연구는 Class Loader와 JIT Compiler를 수정하고 Native Verifier부분을 추가하여서 구현하였다. ClassLoader에서 자바 클래스 파일을 적재할 때, 자바 클래스 파일 내에 네이티브 코드가 있는지 확인을 한다. 네이티브 코드가 존재하지 않는다면 JIT 컴파일 방식과 동일하게 수행된다. 다른 점은 JIT 컴파일러가 네이티브 코드를 생성하면 자바 클래스 파일이 다음 번에 적재될 때를 위해서 대응하는 자바 클래스 파일에 네이티브 코드를 추가한다는 것이다. 그림 1에서 굵은 점선이 새로 추가된 기능이다. 자바 클래스 파일에 네이티브 코드가 존재하면 Class Verifier 대신에 Native Verifier를 이용하여 포함된 네이티브 코드가 올바른 코드인지 검증을 하고 검증에 성공하면 자바 클래스 파일에 포함된 네이티브 코드를 이용한다. 따라서 자바 클래스 파일의 검증 시간과 네이티브 코드로 변환하는 JIT 컴파일 시간을 줄일 수 있다. 네이티브 코드 검증에 실패하면 포함된 네이티브 코드를 제거하고 자바 클래스 파일에 네이티브 코드가 없는 것과 같은 방식으로 수행한다.

3.2 제안하는 네이티브 코드의 디스크 캐쉬 기법

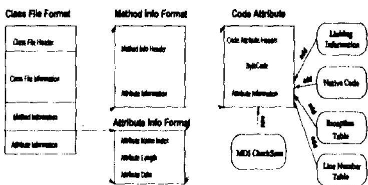

그림 2: 네이티브 코드의 추가

그림 2는 변환된 네이티브 코드를 캐쉬하기 위해서 저장되는 정보와 그 위치를 보여준다. MD5 Checksum은 네이티브 코드의 버전 관리를 위해서 사용되는 정보이다. 3.3절에서 자세히 설명한다. 각 정보의 저장은 자바 클래스 파일의 속성 정보를 이용한다. 본 연구에서는 MD5 CheckSum, Linking Information, Native Code, Exception Table, Line Number Table의 5가지 속성을 정의하고 자바 가상 기계가 처리할 수 있도록 수정하였다.

3.3 제안하는 네이티브 코드의 버전 관리 기법

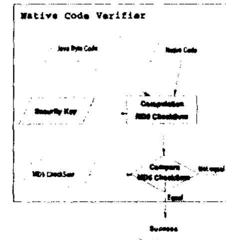

그림 3: 네이티브 코드 검증기

그림 3은 MD5 해쉬 함수를 이용하여 네이티브 코드를 검증하는 방법을 보여준다. 자바 바이트코드와 네이티브 코드를 메시지로 이용하여 하나의 체크섬 값을 구한다. 따라서 자바 바이트코드나 네이티브 코드가 변하면, 저장된 MD5 CheckSum 값과는 다른 체크섬 값이 나오므로 버전 관리가 쉽다. 또한 안전성을 위해서는 Security Key 값을 이용한다. 네이티브 코드나 자바 바이트코드를 수정한 후 새로운 체크섬 값으로 MD5 CheckSum 값을 수정하면, 네이티브 코드 검증기는 이 사실을 알아낼 수 없다. 따라서 네이티브 코드 검증기만 알고 있는 Security Key값을 추가하여 체크섬 값을 구한다. Security Key 값을 알지 못 하는 사용자는 자바 바이트코드나 네이티브 코드를 수정한 후 올바른 체크섬 값을 구할 수 없다. 계산한 체크섬 값이 MD5 CheckSum 값과 다르다면 네이티브 코드 검증기는 검증에 실패한다.

4. 성능 평가

4.1 실험 환경

본 구현은 Kaffe OpenVM[2]를 수정하였다.

- CPU : Intel Pentium II 233Mhz
- L2 Cache : 512 KByte
- RAM : 128MByte
- OS Kernel : Linux-2.2.12
- Local Disk : EIDE UDMA-33MHz

본 논문에서는 SPEC JVM98[1]을 수행하여 실행 시간을 측정하는 방법을 사용하였다.

4.2 초기화 시간 비교

일반적인 동적 컴파일 모델에서 자바 클래스 파일의 초기화 시간은 주로 JIT 컴파일러의 변환 시간에 비례한다. 본 논문의 접근 방법은 JIT 컴파일 시간보다 적은 시간동안에 네이티브 코드를 적재하므로 전체 초기화 시간을 줄일 수 있다. 초기화 시간의 성능 향상은 표 1과 같다. 대략 2배에서 5배 정도의 초기화 시간의 성능 향상이 있으며 평균 2.78배의 성능 향상을 보인다.

Program	Original Kaffe (micro second)	Our Approach	Ratio of Performance
compress	852	379	44%
jess	1808	703	38%
raytrace	1324	454	34%
db	2377	433	18%
javac	4146	1930	46%
mpegaudio	1754	486	27%
jack	2048	825	40%

표 1: 초기화 시간의 성능 향상

4.3 추가 요소

Program	Class (Bytes)	Native Image	Ratio of size
Java Package	587937	184190	0.313
SPEC JVM98	141956	67532	0.476
compress	23598	27649	1.172
jess	44548	241797	5.428
raytrace	33163	148967	4.492
db	25605	25503	0.996
javac	91144	695712	7.633
mpegaudio	38204	406417	10.638
jack	50573	436559	8.632
Total	1036728	2234326	2.156

표 2: 추가 적인 디스크 공간

네이티브 코드를 저장하기 위한 추가적으로 필요한 디스크 공간은 표 2와 같다. 평균적으로 자바 클래스 파일 크기에 **2.146**배 정도 더 필요하다.

4.4 수행 시간과 기대 효과

Program	Original (mili sec)	Our approach	Ratio of performace
compress	73226	72753	1.007
jess	155771	154665	1.007
raytrace	165532	164662	1.005
db	168477	166533	1.012
javac	173184	170968	1.013
mpegaudio	164115	162847	1.008
jack	228712	227489	1.005
Total	1129017	1119917	1.008

표 3: 성능 향상

실제 수행 시간은 표 3과 같다. 평균적으로 0.8%의 성능 향상이 있다. 성능이 좋은 네이티브 코드를 생성하기 위해서는 JIT 컴파일 시간이 더 필요하다. 따라서 본 논문에서 구현한 자바 가상 기계는 JIT 컴파일러의 성능이 좋을 수 록 더 큰 성능 향상을 볼 수 있다.

초기화 시간의 단축으로 가장 큰 효과를 볼 수 있는 분야는 GUI 응용 프로그램이다. GUI 응용 프로그램은 사용자와의 상호 작용이 중요하다. 때문에 자바 클래스 초기화로 인한 프로그램의 일시적인 정지는 체감 속도를 크게 떨어뜨린다. 본 논문에서 구현한 자바 가상 기계는 하나의 자바 클래스 파일을 초기화하는데 평균 **5971μsec** 정도 단축시킨다.

5. 결론

본 논문에서는 디스크 캐쉬 기법을 이용하여 성능을 향상시킨 자바 가상 기계를 구현하였다. 또한 네이티브 코드 검증기를 구현하여 네이티브 코드를 자바 바이트코드 수준의 안전성을 제공하였다.

본 논문에서 구현한 자바 가상 기계의 성능을 SPEC JVM98을 이용하여 측정하였다. 초기화 시간이 2.78배 단축됨을 알 수 있다. 반응 속도가 중요한 GUI 프로그램에서는 체감 속도의 증가가 매우 클 것이다. 또한 현재로서는 전체적인 성능 향상의 폭이 0.8%로 작으나 자바 가상 기계의 성능 향상이 커질 수 록 본 논문에서 구현한 방법의 영향이 커짐을 알 수 있었다. 네이티브 코드를 자바 클래스 파일에 다른 영향 없이 추가함으로써 캐쉬한 네이티브 코드를 관리하는 추가 비용을 없앴다. 네이티브 코드를 캐쉬하기 위해 필요한 추가 공간은 자바 클래스 파일의 2.15배 정도 더 필요함을 보였다.

향후 연구 과제로는 좀더 일반적인 네이티브 검증기의 구현, 그리고 실제적으로 많이 사용되는 GUI 프로그램을 이용하여 구현한 자바 가상 기계의 반응 시간 측정 등이 있다.

참고 문헌

[1] http://www.spec.org/osg/jvm98/.

[2] "What is kaffe open vm?." http://www.transvirutal.com/products/ architecture.html.

[3] J. M. O'Connor and M. Tremblay, "Picojava-i:the java virtual machine in hardware," *IEEE Micro*, vol. 17, pp. 45 53, March/April 1997.

[4] T. Proebsting, G. Townsend, P. Bridges, J. Hartman, T. Newsham, and S. Watterson, "Toba java for applications - a way ahead of time (wat) compiler," in *Proceedings of the Third USENIX COOTS*, pp. 41 53, June 1997.

[5] C.-H. Hsieh, J. Gyllenhaal, and W.-M. Hwu, "Java bytecode to native code translation: The caffeine prototype and preliminary results," in *MICRO-29*, pp. 90 97, December 1996.

[6] G. Muller and U. P. Schultz, "Harissa: A hybrid approach to java execution," *IEEE Software*, vol. 16, pp. 44-51, March/April 1999.

[7] T. Cramer, R. Friedman, T. Miller, D. Seberger, R. Wilson, and M. Wolczko, "Compiling java just in time," *IEEE Micro*, vol. 17, pp. 36 43, June 1997.

[8] http://www.javasoft.com/products/hotspot/whitepaper.html, April 1999. White Paper.

[9] A. Azevedo, A. Nicolau, and J. Hummel, "Java annotation-aware just-in-time (ajit) complilation system," in *Proceedings of the ACM 1999 conference on Java Grande*, pp. 142-151, June 1999.

[10] C.-H. A. Hsieh, M. T. Conte, T. L. Johnson, J. C. Gyllenhaal, and W. mei W. Hwu, "Optimizing net compilers for improved java performance," *IEEE Computer*, vol. 30, pp. 67 75, June 1997.