자바 복합 내장형 시스템을 위한 Just-in-Time 컴파일러

이재목° 김진철 김성무 신진우 정동헌 문수묵
서울대학교 전기컴퓨터 공학부
jaemok° @altair.snu.ac.kr

이상규, 박종목
삼성전자 주식회사[1]

Just-in-Time Compilation for Java Hybrid Embedded Systems

Jaemok Lee°, Jin-Chul Kim, Sung-Moo Kim, Jin-Woo Shin, Dong-Heon Jeong, Soo-Mook Moon
School of Electrical Engineering of Seoul National University.
Sang-Gyu Lee, Jong-Mok Park
Samsung Electronics Co. Ltd.

요 약

내장형 시스템에서 많이 채택되고 있는 자바 가상 머신의 성능을 향상시키기 위해 interpreter, just-in-time 컴파일러 (JITC), ahead-of-time 컴파일러 (AOTC) 세가지 방식을 모두 지원하는 자바 가상 머신을 설계하고 구현하였다. 특히 이런 환경을 지원하기 위한 효율적인 JITC와 시스템의 idle 시간에 JITC 모듈을 활용하기 위한 client-AOTC의 설계와 구현에 대해 살펴보고 현재까지의 실험 결과를 보고한다.

1. 서 론

자바는 최근 휴대폰, 디지털TV, 텔레매틱스 등 많은 내장형 시스템에서 표준으로 채택되고 있다. 이러한 시스템에서 자바가 채택되는 이유로는 우선 가상 머신을 사용함으로써 프로세서, OS, 하드웨어 기반이 다양한 내장형 플랫폼에서 일관된 실행 환경을 제공할 수 있고 보안에 있어서도 바이러스나 해킹 코드로 인한 전체 시스템 붕괴의 가능성이 낮으며, 충분히 성숙된 풍부한 API와 프로그램의 안정성을 높여주는 언어 기능(쓰레기 수집기, 예외 처리)으로 인해 소프트웨어 콘텐츠 개발이 쉽기 때문이다. 이러한 장점에도 불구하고 자바는 " write once, run everywhere" 로 대변되는 강력한 이식성을 보장하다 보니 성능에는 큰 문제점을 안고 있다. 즉 자바는 기계어 코드로 컴파일되는 것이 아니라 바이트코드라는 중간 코드로 컴파일되기 때문에 하드웨어가 아니라 자바 가상 머신 (Java virtual machine, JVM)[1] 이라는 소프트웨어에 의해 해석기(interpreter)를 통하여 수행되고, 따라서 느릴 수 밖에 없다.

이러한 성능 저하 문제를 해결하기 위해서 바이트코드를 기계어 코드로 변환해서 수행하는, just-in-time 컴파일러 (JITC) [2] 나 ahead-of-time컴파일러 (AOTC) [3-5]가 사용되고 있다. 특히, 내장형 시스템에서는 컴파일이 수행시간에 이루어지는 JITC 방식의 컴파일 오버헤드의 부담이 크므로 프로그램 수행이전에 미리 바이트코드를 기계어 코드로 변환하는 AOTC 방식이 많이 사용되고 있으며 국내 휴대폰에서 많이 채택하고 있는 WIPI (http://www.wipi.or.kr/) 에서 자바를 수행하는 방식도 AOTC라고 볼 수 있다.

그러나 AOTC의 경우 이미 내장형 시스템에 정적으로 존재하는 자바 프로그램의 성능향상은 가져올 수 있으나 외부에서 동적으로 다운로드 되는 자바 프로그램의 경우 해석기에 의해 수행되어야 하므로 성능에는 영향을 줄 수 없다. 따라서 이렇게 동적으로 다운로드 되는 자바 프로그램의 성능향상을 위해서는 JVM안에 JITC를 채택하여 성능을 보완하는 것이 바람직하다. 또한 다운로드 된 프로그램들에 대하여 시스템이 수행되지 않는 (idle) 동안에 JITC를 이용하여 미리 컴파일을 해두는 client-AOTC[6, 7]를 사용하는 것이 효율적일 것이다.

본 연구에서는 해석기, AOTC, JITC, client-AOTC가 복합적으로 사용되는 내장형 자바 환경의 구조를 제안하고 여러 가지 설계문제들에 대하여 해결책을 제시한다. 특히 효율적인 JITC의 설계와 구현에 있어서 이슈가 되는 부분들에 대한 선택 사항들을 기술한다. 본 연구 결과는 Sun Microsystems 사의 J2ME CDC환경의 자바 가상 머신인 CVM[8]에 구현하였으며 현재까지의 JITC의 성능 결과를 보고 한다.

이 논문의 구성은 다음과 같다. 2장에서는 제안된 AOTC/JITC/client-AOTC/ 복합 자바 수행 환경 구조를 설명하고 3장에서는 JITC 및 Client-side AOTC의 설계와 구현을 기술한다. 4장에서는 현재까지의 시험결과를 보고하고 5장에서 요약한다.

[1] 본 연구는 삼성전자의 지원을 받아 수행되었다.

2. 복합 자바 수행 환경의 구조

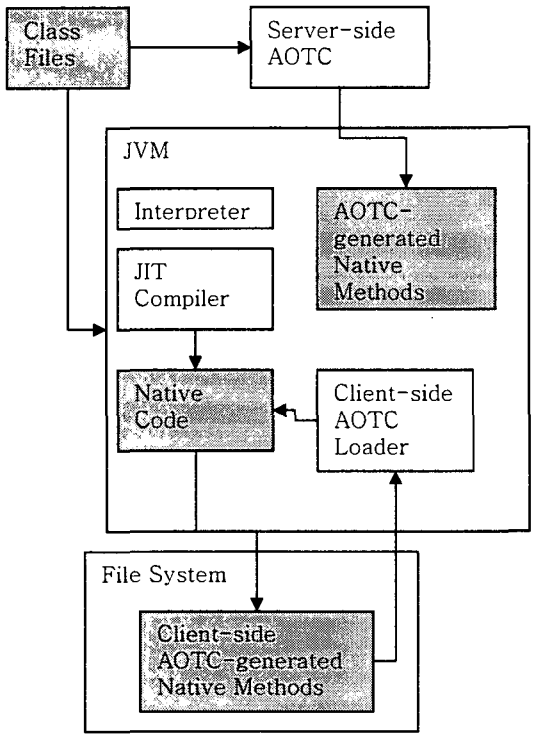

그림 1 복합 자바 수행환경의 구조

그림 1은 본 연구에서 개발한 복합 자바 수행환경의 구조를 표현하고 있다. 해석기와 JIT 컴파일러가 공존하고 있다.
JVM을 빌드하는 과정에서 일부 라이브러리 메소드들은 Java-to-C 방식의 AOT 컴파일러(Server-side AOTC)를 이용하여 기계어코드로 JVM 실행파일에 내장시킨다.
Server-side AOTC 는 JIT 컴파일에 걸리는 시간과 메모리 사용량을 줄이는 동시에, 보다 효과적인 최적화를 통한 성능향상을 가져올 수 있다.
JVM 수행 과정에서, JIT 컴파일러가 생성한 기계어 코드는 재배치 정보를 추가하여 파일시스템에 저장하고, 필요에 따라 본 연구에서 개발한 loader 를 이용해서 메모리에 적재하여 사용할 수 있다.
Client-side AOTC 는, JIT 컴파일 된 결과를 파일로 저장함으로써, 시스템이 재시작 했을 경우, JIT 컴파일을 하지 않고, 바로 파일에 있는 내용을 읽어서 사용할 수 있어, JIT 컴파일에 걸리는 시간을 줄일 수 있으며, 다운로드 받은 클래스파일을 시스템의 휴지시간동안, 미리 컴파일해 놓음으로써 시스템의 효율을 향상시킬 수 있다.

3. JITC의 설계 및 구현

그림 2는 본 연구에서 개발한 JIT 컴파일러의 컴파일 과정을 그린 것이다
바이트코드 메소드를 입력받아 메소드 인라인 최적화를 하고, 그 결과를 바탕으로 중간코드를 생성한다.
생성된 중간코드는 Tree-region 에서 CSE[9], LICM[10], Constant Propagation[11], Exception Check Elimination[12] 최적화를 거쳐 최적화된 중간코

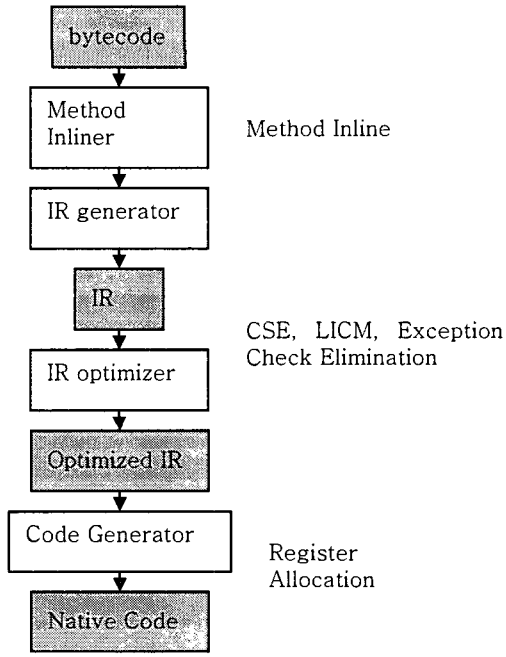

그림 2 JIT 컴파일러 구조

드로 변환된다. 그리고 마지막으로 레지스터 할당을 통해 기계어코드가 생성된다.

3.1 중간코드 설계

중간코드를, 기계어 수준으로 세부적으로 설계할 경우, 최적화를 할 수 있는 기회는 많아지지만, 컴파일 도중의 메모리 요구량이 많아지고, 컴파일 시간도 길어진다.
반면 중간코드를 바이트코드+피연산자 수준으로 설계할 경우, 중간코드가 간단해지기 때문에 메모리 요구량이나 컴파일 시간 면에서는 효율적이지만, 최적화 기회를 많이 놓칠 수 밖에 없다.
본 연구에서는, 프로그램의 정상적인 흐름에 관계가 없는 예외검사코드들과 기계어 수준으로 세분할 경우, 지나치게 복잡해지는 명령어들, 즉 메소드 호출, 모니터처리 등의 명령어들은 가능한 간단한 중간코드로 생성하고, 그외의 연산에 관련된 명령어들은 세부적인 중간코드로 생성함으로써 보다 많은 최적화 기회와 함께 효율적인 컴파일이 가능하도록 하였다.

3.2 레지스터 할당

본 연구에서는 다른 JIT 컴파일러에서도 많이 사용하는 tree region 을 레지스터 할당의 단위로 사용한다.
Tree region 을 단위로 사용하는 것은, 그 안에서는 predecessor 의 개수가 1개뿐이기 때문에, predecessor basic block 에서 할당된 결과를 successor basic block 이 그대로 이용할 수 있기 때문에 레지스터 할당이 간단하고 빨라진다는 장점이 있다.
대신, tree region 들 사이에 레지스터 할당 결과를 맞추어 줘야 하는데, 이 과정에서 필요없는 레지스터 spill/reload 가 발생할 수 있다.
LaTTe [13] 에서는 레지스터 spill/reload 로 tree-

region 사이를 연결해주는 방식 대신 copy 명령어를 삽입하는 방식으로 이 문제를 해결하였다. 그리고, backward sweep 을 통해 이러한 copy 명령어를 많이 줄일 수 있었다.
본 연구에서도 LaTTe 와 같은 copy 명령어를 이용한 register reconcile 을 사용하고 있다. 하지만, backward sweep 을 사용하는 대신, sibling 으로 레지스터 할당결과를 전달하여 참고하는 방식을 사용한다.
sibling 들이 서로 비슷한 레지스터 할당 결과를 갖고 있으면, 그것들의 공통된 successor basic block 에 대해서 register reconcile 을 할 필요가 적어지기 때문이다.
Backward sweep 은 register reconcile 을 위한 copy 명령을 줄이기 위한 것도 있지만, calling convention을 맞추기 위해 필요한 copy 명령어를 줄이는 것이 큰 이유였는데, mixed mode 에서는 native function call 방식을 사용하지 않기 때문에, 더 간단한 sibling 으로 레지스터 할당결과를 전달하는 방식을 사용하였다.

3.3 Mixed-mode 를 위한 설계 및 구현

본 연구에서는, 여러가지 자바 수행 방식을 동시에 지원하고 있기 때문에, 이러한 방식들 간의 호환성을 유지해주는 것이 큰 문제 중의 하나이다.
인터프리터와 함께 수행되며, 정확한 쓰레기 수집(precise garbage collection)을 하는 환경에서는 인터프리터에서 사용하는 자바 스택 프레임을 JIT 과 AOTC 에서도 같은 방식으로 사용하도록 하는 것이 가장 간단하다.
LaTTe와 같은 JIT 컴파일러에서는 연결리스트 자료구조로 구성된 자바 스택 프레임을 사용하지 않고, 기존의 native 프로그램들이 사용하는 스택 프레임을(sp 레지스터를 사용하는 방식) 사용하였다. 이 방식은 매번 함수 호출때마다 스택 프레임을 할당하기 위해서 현재 스택프레임의 상태나 용량을 체크할 필요없이 sp 레지스터 값만 변경해주면 된다. 하지만, 인터프리터와 함께 수행되는 환경에서는, 자바 스택프레임과 native 스택프레임이 번갈아가며 공존할 수 있기 때문에, 스택프레임들간의 연결구조를 유지해줘야 하며, 결국, native 스택프레임을 사용하는 이점을 살리기 힘들다.

3.4. Client-side AOTC의 설계 및 구현

Client-side AOTC 는 두가지의 추가적인 모듈로 구성된다. 첫번째는, JIT 컴파일러가 생성한 코드를 파일시스템에 저장하는 모듈이고, 두번째는 파일시스템에 저장된 코드를 읽어서 코드 재배치를 한 뒤에 메모리에 적재하는 모듈이다.
JIT 컴파일러가 생성한 코드는 직접함수호출, 정적변수 접근, 시스템 클래스 정보 접근, 전역변수 접근 등 코드 외부에 존재하는 변수나, 함수들에 접근하는 코드들을 포함하고 있는데, 이러한 변수나 함수들의 주소는 매번 JVM 을 수행하면서 그 위치가 변할 수 있다.
그래서 이러한 코드들의 위치와 해당 코드가 접근하려고 하는 개체의 symbolic 정보를 함께 파일시스템에 저장하고, 메모리에 적재시에는, 이러한 정보를 이용하여 코드를 변경해주어야 한다.

4. 실험결과

그림 3은, MIPS architecture 에서 SPECjvm98[15] 의 프로그램들을, JIT 컴파일을 켜고 인터프리터와 함께 수행했을 때, 인터프리터만으로 수행했을때와 비교해 수행시간이 얼마나 줄어들었는지를 나타내고 있다. 비교대상은 CDC RI 의 인터프리터로, threading 기법[14]을 사용하여, 바이트코드 해석에 필요한 부하를 많이 줄여놓은 상태이며, JIT 컴파일 도중, mixed-mode 를 위해 보완적으로 생성된 코드들 때문에 성능 향상이 종래의 JIT 컴파일러에 비해 크지는 않다.

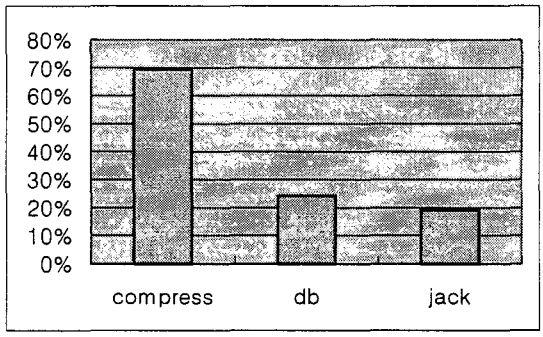

그림 3 Interpreter 대비 성능향상

5. 요약

본 연구에서는, 인터프리터와 JIT 컴파일러, Server-side AOTC, Client-side AOTC 등 여러가지의 자바수행 방식이 함께 수행될 수 있는 mixed-mode JVM 을 설계, 개발하였으며, 그 성능을 측정하였고, SPECjvm98 벤치마크에서 20~70% 의 성능향상을 확인하였다.

6. 참고문헌

1. T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Addison Wesley, 1997
2. A.-R. Adl-Tabatabai, M. Cierniak, G.-Y.Lueh, V. M. Parikh, and J. M. Stichnoth. *Fast, effective code generation in a just-in-time Java Compiler*. In Proceedings of the ACM SIGPLAN '98 Conference on Programming Language Design and Implementation, pages 280-290, 1998
3. Jove: super optimizing deployment environment for Java, http://www.instantiations.com/_NewSite/jove/JOVEReport.pdf
4. Robert Fitzgerald, et al., *Marmot: An Optimizing Compiler for Java,* Software Practice and Experience, **30**(3) p, 199-232, 2000
5. Mauricio Serrano, et al., *Quicksilver: a quasi-static compiler for Java,* ACM SIGPLAN Notices, **35**(10) p, 66-82, 2000
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. CDC Foundation Porting Guide, http://java.sun.com
9. Cocke, J. *Global common subexpression elimination.* SIGPLAN Notices 5, 7 (July 1970), 20-24
10. Karl-Heinz Drechsler, Manfred P. Stadel, *A variation of knoop, Ruthing, and Steffen's Lazy Code Motion*, ACM SIGPLAN Notices, v.28 n.5 p.29-38, May 1993
11. Wegman, M. N., Zadeck, F.K. *Constant propagation with conditional branches.*ACM Trans. Program. Lang. Syst. 13,2, 181-210. 1991
12. Seungil Lee; Byung-Sun Yang; Suhyun Kim; S. Park; Soo-Mook Moon; K. Ebcioglu; E. Altman, *Efficient java exception handling in just-in-time compilation.* Proc. of ACM Java Grande 2000 Conference, Jun. 2000
13. LaTTe: A Java VM Just-in-Time Compiler with Fast and Efficient Register Allocation, http://latte.snu.ac.kr
14. Anton M. Ertl, *A portable forth engine.* http://www.complang.tuwien.ac.at/forth/threaded-code.html
15. SPECjvm98. http://www.spec.org/jvm98