자바바이트코드의 네트워크 기반 프로파일링 시스템 설계 및 구현†

정 준 영*, 정 민 수*, 이 은 정**, 정 연 정**, 윤 기 송**
*경남대학교 컴퓨터공학과,
**한국전자통신연구원

Design and Implementation of Network-based Profiling System of Java Bytecode

Jun-Young Jung* , Min-Soo Jung* , Eun-Jung Lee** , Yeon-Jeong Jeong** , Ki-Song Yoon**
*Dept. of Computer Engineering, Kyungnam University,
**Electronics and Telecommunications Research Institute

요 약

네트워크상에 바이트 코드로 존재하고 실행되는 자바 프로그램은 여러가지 컴퓨팅 환경 요소로 인해 수행병목현상이 나타나고 있다. 더구나 프로그램 성능 향상을 위해 바이트코드를 해석한다는 것은 쉬운 일이 아니다. 본 논문은 자바 바이트코드를 소스코드로 디컴파일하고, 프로그램 문맥의 기본 블록과 제어 구조를 분석하여 프로파일 정보를 얻는다. 그리고 프로파일 정보를 바탕으로 사용자나 개발자에게 의미있는 정보를 제공하여 최적화된 프로그램을 개발할 수 있도록 하는 네트워크 기반 프로파일링 시스템과 프로파일 정보 시각화를 위한 시스템의 프로토타입 설계 및 구현을 다룬다.

1. 서 론

자바 프로그램의 성능은 가상기계의 성능과 프로그램의 문맥에 따라 좌우된다. 사용자나 개발자들은 최적의 프로그램을 개발하기 위해 바이트코드의 수행을 평가하는데 관심을 기울여왔다. 자바소스코드는 바이트코드로 컴파일 되고 가상기계상에서 운영되므로 실행속도가 느리다. 또한 일정치 않은 기계의 프로세서 속도와 네트워크 속도, 대역폭과 connection(통신에서 두 장치가 같은 채널을 사용하려는 상황을 말함)은 수행에 있어서 병목 현상을 초래할 수 있다.

본 논문은 사용자나 개발자들이 병목 현상이 발생하는 위치를 개선하여 최적의 프로그램을 개발할 수 있도록 하는데 주안점을 둔다. 사용자 관점에서는 바이트코드를 해석한다는 것은 쉽지 않기 때문에 실행병목을 식별하는 것은 불가능할 수도 있다. 따라서 설계하고자 하는 시스템은 네트워크를 기반으로 인터넷을 이용하여 동작한다. 바이트코드를 이용하여 프로파일 정보를 추출해 내고 그 프로파일 정보를 바탕으로 사용자나 개발자에게 의미있는 정보를 시각화 시켜 제공함으로써 프로그램을 효율적으로 개발하는데 활용하도록 한다.

본 논문의 구성은 다음과 같다. 2장에서는 네트워크 기반의 프로파일링 시스템에 대한 소개와 시스템을 구성하는 각 컴포넌트의 기술적인 항목, 그리고 구현에 앞서 고려할 사항을 기술한다. 3장에서는 프로파일링 시스템의 구현 논점을 기술한다. 4장에서는 결론 및 활용 방향을 제시한다.

2. 바이트코드의 네트워크 기반 프로파일링 시스템

자바 바이트코드의 네트워크 기반 프로파일링 시스템은 바이트코드를 입력으로 받아 프로그램의 기본 블록과 제어구조를 분석하여 결과를 시각적으로 보여주기 위한 도구이다.

2.1 시스템의 전체 구조

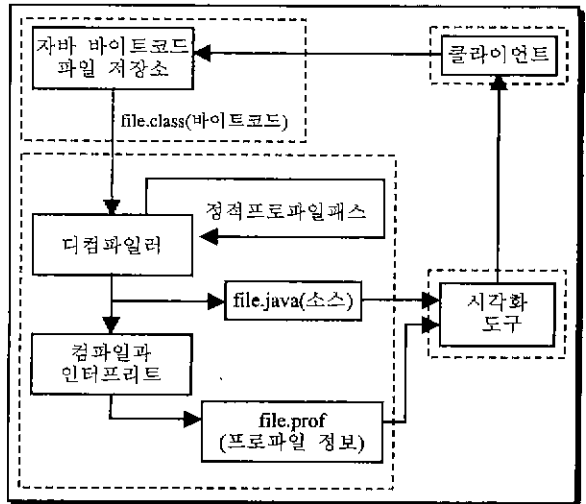

그림 1. 논리적인 시스템 구조

그림 1 의 시스템 구조에서 점선으로 강조된 부분들은 시스템에서 독립적인 논리 개체이며, 이들 개체는 네트워크에서 서로 다른 노드에 물리적으로 존재한다.

클라이언트는 네트워크를 통해서 프로파일 하려는 바이트코드(file.class)가 존재하는 노드에 연결한다. 브라우저를 이용해서 툴을 로드하고 프로파일을 위한 바이트코드를 선택한다. 선택된 바이트코드는 소스코드를 복원하는 디컴파일러의 입력파일로 취해지며 프로그램 기

† 본 연구는 한국전자통신연구원의 연구 개발 지원금에 의해 작성되었습니다

본 블럭과 제어 구조의 정보를 가지는 고수준 소스(file.java) 파일을 생성한다. 고수준 소스 파일은 프로파일 정보(file.prof) 파일을 생성하기 위해 컴파일 되고 인터프리트 된다. 고수준 소스 파일과 프로파일 정보 파일은 클라이언트에게 프로파일 정보를 보다 쉽게 이해하도록 고수준 소스파일에 따라 계층적으로 프로파일 정보를 나타내는 시각화 도구(visualization tool)의 입력 파일로 사용되며 수행은 클라이언트 노드에서 이루어진다.

▶ 디컴파일러(reverse compiler)
바이트코드를 읽어들여 JavaIR 이라는 일반 고수준 중간 표현물로 변환한다. JavaIR 은 소스코드가 생성되기 전에 다중패스에 의해 수정될 수 있으며 신뢰성 있는 자바소스를 생성하기 위한 작업이다.

▶ 프로파일러(profiler)
정적 패스(static pass)와 런타임 라이브러리(runtime library)로 구성된다.
(1) 정적 패스 : 필요한 선언과 런타임 프로파일러 라이브러리와 관련한 추상문법(abstract syntax)가 디컴파일러에 의해 생성되고 난 후에 복원된 소스파일을 생성하기 위해 마지막 호출이 발생하기 전에 수행된다. 다음과 같은 다양한 블록을 삽입한다.
-루프 : 런타임 프로파일러(runtime profiler) 호출전과 루프의 종료 후에 삽입된다. 루프에서 반복횟수도 삽입된다.
-메소드 : 메소드의 시작과 종료(return 문 전)에서 런타임 프로파일러 호출이 삽입된다.
-메소드 호출 : 모든 메소드 호출 전후에 삽입된다. 만약 메소드가 프로파일되지 않았다면 메소드에 대한 총 실행 횟수를 반환한다.
-if-then-else 문 : 호출은 then 과 else 절의 시작에 삽입된다.
(2) 런타임 라이브러리 : 정적 패스에 의해 삽입된 호출과 선언을 이용해 기본 블록에 대응하는 아래와 같은 이벤트를 기록한다.
- While 루프 : 각 루프의 반복횟수와 합계를 기록한다.
- If-then-else 문 : if 와 else 의 수를 기록한다.
- 메소드 : 메소드 실행의 총횟수, 호출과 해제의 횟수를 기록한다.

▶ 시각화 도구(Visualization Tool)
사용자가 특정한 객체 인스턴스나 메소드 인스턴스에서 호출되는 프로파일 정보를 이해하기 쉽도록 디컴파일된 소스에 따라 시각적으로 표시한다.
(1) 입력처리(Input processing)
런타임 시스템(runtime system)에 의해 생성된 프로파일 정보와 정적 패스(디컴파일러+프로파일러)로부터 복원된 소스코드를 전처리(preprocessing)한다.
(2) 초기화 디스플레이(Initial display)

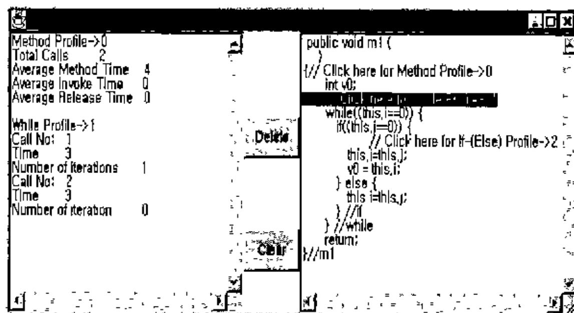

그림 2. 초기화 디스플레이

프로파일 캐쉬(cache)와 디컴파일된 소스를 구성한다. 그림 2 는 유용한 프로파일 정보를 생성하기 위해 주석과 함께 선택되어질 수 있는 소스의 행을 나타낸다. 그림 3 은 그림 2 의 선택적인 행을 클릭함으로써 팝업되는 다이얼로그 박스이다.
(3) 프로파일 윈도우 (Profile Window)
행의 하나를 클릭함으로써 왼쪽부분에서 선택할 수 있는 인스턴스와 인스턴스 요약(instance summary)들의 목록을 얻엔다. 사용자에 의해 클릭되어짐으로써 관련된 요소가 선택되면 프로파일 정보를 나타낸다.

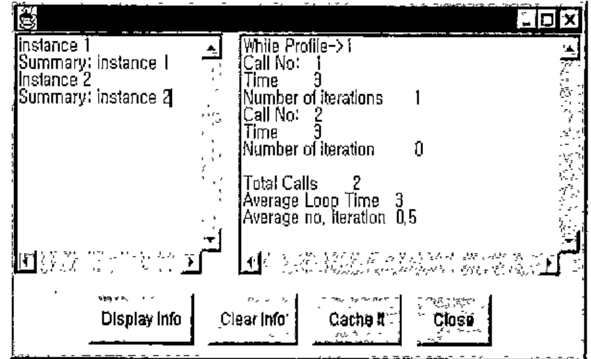

그림 3. 프로파일 윈도우

2.2 네트워크 기반 프로파일 시스템 구현을 위한 고려 사항

자바 인터프리터가 내장된 현재의 웹브라우저는 강력한 보안으로 인해 동적 클래스 수정을 지원하지 않으므로 네트워크를 통해서 코드를 설치할 수 없다. 그리고 그림 1 에서 보는 바와 같이 설계 및 구현하려는 시스템에서는 독립 논리 개체를 구분할 수 없다.
본 논문에서는 다음과 같은 방법을 제안한다. 프로파일하기 위한 바이트코드와 프로파일러는 같은 호스트에 존재한다. 클라이언트 요구로 바이트코드는 역으로 컴파일되고, 정적 프로파일러 패스(static profiler pass)가 삽입된다. 그리고 바이트코드는 런타임 프로파일링 시스템(runtime profiling system) 호출과 함께 소스코드로 복원된다.
서버로의 연결은 애플릿이 호스트상에 알려진 포트번호에 연결함으로써 이루어진다. 연결이 이루어지면 애플릿은 프로파일하기 위한 바이트코드 이름을 전달한다. 프로파일과 고수준 소스코드는 인터넷을 통해 접근 가능한 위치에 생성된다. 시각자(visualizer)는 클라이언트에 의해 요구된 프로파일 정보와 요약 정보를 보여준다. 시각화하는 단계는 두개의 입력파일이 생성되어져서 접근 가능하기만 하면 서로 다른 서버상에서 다른 단계와 독립적으로 사용될 수 있다.

3. 구현 논점

3.1 디컴파일러

입력으로는 바이트코드를 사용한다. 왜냐하면 소스가 없는 응용 프로그램(애플릿)이 네트워크상에 분산되어 존재하기 때문이다. 프로파일된 출력은 시각화하는 단계에 제공되어야 한다. 디컴파일러는 다중 전단부(multiple front-end)를 포함하는 컴파일러를 사용한다. 전단부는 JavaIR 이라는 고수준 중간 표현물을 만든다. 후

단부(back-end)는 지원되는 목적언어와 같은 JavaIR을 작성할 수 있다.

3.2 프로파일러

프로파일 데이터는 프로파일되는 프로그램의 정상적인 실행을 위해 시간과 기억 공간 요구를 제외하고는 거의 오버헤드를 발생시키지 않는 방법으로 구성되어야 한다. 프로파일 데이터를 모으는 실행시간 오버헤드는 정상적인 프로그램 연산이 가능하도록 충분히 작아야 한다.

모든 프로파일/프로파일 요약을 임의적으로 관리 유지한다고 하면 프로파일 정보는 객체 인스턴스에 저장된다.

▶ 정적 단계(static phase)

그림 4는 프로파일러 패스로 바이트코드 파일을 역 컴파일한 코드이다. 프로파일러 패스로 추가된 명령은 키워드 "profile"로 식별할 수 있다. 각 클래스 인스턴스는 형 프로파일(type profile)의 객체와 관련되어 있다. 관계 이벤트는 while 루프의 시작이나 if 분기로 취해지는 부분처럼 발생하고, 관계 프로파일 메소드에 대한 호출이다. 이들 메소드에 대한 매개변수는 프로파일된 각 블록에 대한 정적 참조(static reference)이다. 기본 블록은 코드에서 나타날 때마다 번호가 매겨진다.

프로파일 객체(profile object)의 인스턴스화는 클래스의 이름과 클래스에서 프로파일된 기본 블록의 수를 매개변수로써 가진다.

```
Class loops1 extends java.lang.Object
{
   public int i ;
   public int j;
   public Rprofile profile = new Rprofile(7,"loops1"),

   public void m1() {
     int v0;
     profile.start_profile_method(0),
     profile.before_while_invoke_time(1);
     while((this.i ==0)) {
       profile.while_count(1);
       if((this.j ==0)) {
         profile.if_profile(2);
         this.i = this.j;
         profile.capture_invoke_time(3),
         global.m2();
         profile.capture_release_time(3);
         v0 = this.i;
       }else{
         profile.else_profile(2);
         this.i = this.j;
       } //if
    } //while
    profile.after_while_invoke_time(1);
    profile.end_profile_method(0);
    return;
  } //m1
}
```

그림 4. 정적단계의 예

▶ 런타임 라이브러리

런타임 라이브러리는 Rprofile 클래스로 구성되어 있다. Rprofile class는 클래스의 이름과 프로파일된 기본 블록의 수, 즉 두개의 매개변수로 설명된다. 이들 매개변수는 private 변수에 저장되고, 상응하는 프로파일 유닛(profile unit) 수를 나타낸다. 프로파일 유닛(while, if-then-else, method)에 대한 첫번째 호출이 있을 때, 프로파일 유닛형은 초기화되고, 코드에서 기본 블록과 관계있는 이벤트를 저장하기 시작한다.

- while 루프에서는 시작에서 완료까지 호출마다 반복되는 총 수로써 시간을 측정한다.
- if-then-else 문에서 then 분기가 취해지면 if 카운트를 갱신하고 else 분기가 취해지면 else 카운트를 갱신한다.
- 메소드에서는 시작에서 완료까지 메소드 호출시간과 해제시간으로써 시간을 측정한다.

3.3 시각화 도구

시각화 도구은 사용자가 관계있는 코드 부분을 클릭할 수 있고, 관심있는 프로파일 정보를 볼 수 있도록 GUI를 생성한다. 이것은 정적 단계의 출력인 file.java와 런타임 단계의 출력인 file.prof를 전처리한다. file.java의 전처리는 file.java 내에 있는 스트링의 리스트, 분석된 프로파일러 관계 코드, 그리고 주어진 리스트 요소를 선택할 수 있는지 나타내기 위한 플래그(flag) 삽입 과정이다. file.prof 전처리는 정보를 데이터 구조에 저장하는 과정이다.

사용자가 관련 프로파일 리스팅을 선택하면, Profile_Window 호출을 통해 다이얼로그 박스를 생성한다. 사용자는 몇가지 옵션을 통해서 프로파일 정보를 볼 수 있다. 사용자가 display 버튼을 클릭해서 선택하면, 모든 선택된 정보는 편집할 수 없는 텍스트영역에 나타난다.

4. 결론 및 연구 결과의 활용

자바 프로그램의 성능은 자바가상기계의 성능과 프로그램의 문맥에서만 좌우되는 것이 아니라 네트워크 환경에 따른 병목현상에 의해서 좌우될 수도 있다.

본 논문에서 설계한 "네트워크 기반 프로파일링 시스템"은 자바바이트코드를 분석하여 자바의 성능 문제를 파악하고, 이를 해결하기 위한 분석 자료로 제공한다. 또한 분석 자료를 계층적으로 쉽게 파악할 수 있도록 시각화 메커니즘을 제공한다. 프로그램의 개발자나 사용자는 프로그램을 효율적으로 개발하는데 프로파일 정보를 활용할 수 있을 것이다.

참 고 문 헌

[1] B. Veners, *Inside The Java Virtual Machine*, McGraw-Hill, 1998

[2] T. Ball and J. R. Larus. Optimally Profiling and Tracing Programs. In *ACM TOPLAS*, July 1994.

[4] A. D. Samples. Profile Driven Compilation. In *PhD Thesis, UC-Berkeley, UCB-CSD-91-627.*, 1991.

[5] K.Pettis and R. C. Hanson. Profile Guided Code Positioning. In *SIGPLAN Notices ACM* (25), *PLDI*, June 1990.

[6] Coutant and others. Measuring the Performance and Behaviour of Icon Programs. *IEEE Transactions on Software Engineering.*, SE-9(1), January 1993.

[7] 류동항, 정민수, "자바 바이트 코드 분석기의 설계 및 구현" 한국정보과학회 '98 봄 학술발표논문집(A), 제 25 권 제 1 호, pp. 77~79, 1998

[8] 이종동, 정민수, "자바 메소드 호출 관계 표시기의 설계 및 구현" 한국정보과학회 '98 봄 학술발표논문집 (A), 제 25 권 제 1 호, pp. 74~76, 1998

[9] 이종동, "자바 가상 기계 프로파일러의 설계 및 구현", 경남대학교 대학원 석사논문, 1998.