코드 삽입을 이용한 힙 사용량 분석기

주성용° 조장우
동아대학교 컴퓨터공학과
jheaven1@donga.ac.kr, jwjo@dau.ac.kr

Heap Consumption Analyzer Using Code Embedding

Seongyong Joo° , Jang-wu Jo
Dept. of Computer Engineering, Dong-a University

요 약

프로그램의 힙 사용량이나 수행 시간과 같은 프로그램의 동적인 속성을 분석하기 위해서 프로파일러가 이용된다. 자바에서는 가상기계와 프로파일러의 통신을 목적으로 JVM TI 같은 프로파일러를 위한 인터페이스를제공한다. 그러나 자바 가상기계 구현 명세는 프로파일러 인터페이스 구현을 요구하지 않는다. 프로파일러 인터페이스를 구현하지 않는 자바 가상기계에서는 JVM TI를 사용하는 프로파일러를 이용할 수 없다. 본 논문에서는 프로파일러 인터페이스를 사용하지 않는 힙 사용량 분석 기법을 제안했다. 이 방법은 힙 사용 정보를 추출하기 위한 코드를 소스파일에 삽입한다. 이 방법은 힙 사용량 분석 시 자바에서 제공하는 인터페이스를 사용하지 않기 때문에, 표준 인터페이스를 구현하지 않는 가상기계에서도 힙 사용 정보 분석을 수행할 수 있다.

1. 서 론

프로파일러는 프로그램의 성능 분석을 위해서 사용된다. 특히 메모리 사용량이나 수행 속도와 같이 정적으로 계산하기 힘든 속성들을 평가하는데 많이 사용된다[1].

자바는 프로파일러를 지원하기 위한 표준 인터페이스를 제공하고 있다. 현재 자바 1.5버전에서는 프로파일러를 위한 표준 인터페이스로JVM TI를 지원하고 있고, 이전 버전에서는 JVMPI를 제공하고 있다[2][3]. JVM TI를 이용한 프로파일러로는 HProf과 JProfiler 그리고 JBossProfiler 등이 있고, NetBeans나 Eclipse 같은 통합개발환경에서도 이 같은 인터페이스를 사용하는 프로파일러를 지원한다[4][5][6][7].

그러나 프로파일링을 위한 표준 인터페이스 지원은 자바 가상기계 명세의 요구사항은 아니다[8]. 그러므로 모든 자바 가상기계가 프로파일러 인터페이스를 지원하는 것은 아니다. 특히 J2ME와 같이 소형 메모리 환경을 대상으로 하는 플랫폼에서는 이 같은 인터페이스를 지원하지 않는 경우가 있다. 이런 경우 표준 인터페이스를 사용하는 프로파일러는 사용할 수 없다.

본 논문에서는 위에서 기술한 문제를 해결하기 위해서 표준 인터페이스를 사용하지 않는 힙 사용량 분석 기법을 제안한다. 이 기법은 프로그램에서 사용하는 힙 사용량을 계산하기 위해서 대상 프로그램의 소스 파일을 분석하고, 소스 파일이 힙 상태를 변경하는 구문을 포함한다면 힙 사용 정보를 추출하기 위한 코드가 삽입된다. 삽입된 코드를 포함하는 프로그램은 자바 가상기계에서 실행되면서 힙 사용 정보를 추출하고 보고한다. 이 기법에서는 정확한 힙 사용량 분석을 위해서 쓰레기 객체를 고려한다.

본 논문에서 제시하는 기법의 장점은 자바에서 제공되는 표준 프로파일러 인터페이스를 사용하지 않고 힙 사용량을 분석하기 때문에 프로그램이 실행될 수 있는 모든 가상기계 상에서 동작한다는 점이다. 그리고 단점은 힙 사용량을 정확한 수치로 표현할 수 없다는 것이다. 그 이유는 객체 표현 방식은 가상기계 구현에 의존하기 때문이다.

본 논문의 구성은 다음과 같다. 2절에서는 연구 배경에 관해서 기술하고, 3절에서는 코드삽입을 이용한 힙 사용량 분석의 개요에 대해서 설명하겠다. 그리고 4절에서는 실험 결과를 기술하고, 마지막으로 5절에서 결론을 맺는다.

2. 연구 배경

프로그램 실행 중 사용되는 힙 사용량을 분석하기 위한 기존의 방법들은 프로파일러를 사용하는 경우가 많다. 자바는 프로파일링을 지원하기 위해서 JVM TI와 같은 표준 인터페이스를 제공한다. 그림 1은 JVM TI를 사용하는 프로파일러의 동작방식이다.

그림 1에서 Application은 힙 사용량을 분석을 위한 대상 프로그램이고, JVM은 Java 가상기계이다. JVM TI는 표준 프로파일러 인터페이스를 구현한 것이고, Agent는 JVM TI를 통해서 가상기계로부터 관심 있는 정보를 추출하고 Viewer와 통신한다. Viewer는 Agent가 추출한 정보를 보여주는 프로그램이다.

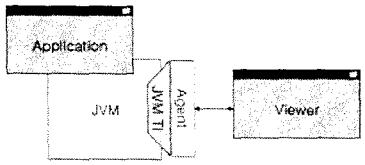

그림 1. JVM TI를 사용하는 프로파일러 동작 방식

이 방법은 대상 프로그램이 실행되고 있는 가상기계에서 사용하는 메모리 정보를 보여주기 때문에 가상기계에 의존적인 속성들을 고려한다는 장점이 있다.

그러나 프로파일러 인터페이스 지원은 자바 가상기계 구현 명세의 요구사항이 아니다. 만일 가상기계가 프로파일러 인터페이스를 지원하지 않는다면 이런 인터페이스를 이용하는 프로파일러를 사용할 수 없다. 이 같은 문제를 해결하기 위해서는 프로파일러 인터페이스를 사용하지 않는 프로파일링 기법이 요구된다.

본 논문에서는 힙 사용 정보를 추출하는 코드를 소스 파일에 삽입함으로써 이 같은 문제를 해결한다. 이 방법은 대상 프로그램의 소스 파일에 힙 사용 정보를 변경하는 구문이 존재한다면 그 구문에 힙 사용 정보 추출을 위한 코드를 삽입한다. 코드가 삽입된 프로그램은 자바 가상기계에서 실행되면서 힙 사용 정보를 추출한다. 그림 2는 코드 삽입을 이용한 힙 정보 분석 기법의 동작방식이다.

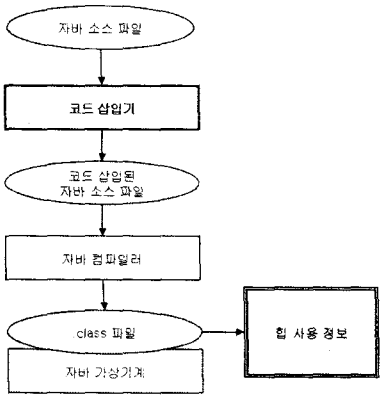

그림 2. 코드 삽입기를 이용한 정보 추출 동작 형태

그림 2에서 코드 삽입기는 자바 언어로 작성된 소스 파일에 힙 사용 정보를 추출하기 위한 코드를 삽입하는 도구이다. 코드 삽입된 자바 소스 파일은 자바 컴파일러에 의해서 클래스 파일로 변환되고, 이 파일은 가상기계에서 실행되면서 필요한 힙 정보를 수집하고, 그 결과를 보고한다.

3. 코드삽입을 이용한 힙 사용량 분석의 개요

힙 사용량 계산을 위한 가장 단순한 방법은 new 연산자에 의해 할당된 힙 사용량을 합산하는 것이다. 그러나 이 방법은 쓰레기 객체를 고려하지 않기 때문에 계산 결과는 쓰레기 객체에 할당된 힙 사용량을 포함한다. 정확한 힙 사용량을 계산하기 위해서는 쓰레기 판별이 요구된다. 쓰레기 판별을 위해서 각 객체들을 참조하는 변수들의 정보를 유지해야 한다. 힙 사용 정보를 변경하는 구문으로는 다음과 같은 5가지가 있다.

① new 연산자에 의한 새로운 객체할당
② 배정문에 의한 객체 참조 변경
③ 메서드 인수로서 참조 전달
④ 메서드 반환 값으로 참조 전달
⑤ 블록 구조

위 다섯 가지 구문을 설명하기 위해서 힙을 리스트로 표현하였다. 힙 리스트의 각 노드는 튜플로 표현하고 이 튜플은 인스턴스의 이름과 그 인스턴스를 참조하는 변수들로 구성된다. 예를 들면 다음과 같다.

- 힙에 인스턴스 obj가 할당되어 있고, 이 인스턴스를 참조변수 var1과 var2가 참조하는 경우

  Heap:[<obj, var1, var2>]

- 힙에 obj1과 obj2가 할당되어 있고, 이 obj1을 참조변수 var1이 참조하고 obj2를 var2가 참조하는 경우

  Heap:[<obj1, var1>, <obj2, var2>]

- 힙 사용 정보가 변경되는 경우, 변경 전·후 상태는 힙에 아래첨자를 기술해 표현

  $Heap_{pre}$:[<obj1, var1>,<obj2, var2>]

  $Heap_{post}$:[<obj1, null>,
  <obj2, var2, var1>]

이것은 처음 힙에서는 참조변수 var1이 obj1을 참조하다가 obj2로 참조가 변경되는 상태를 보인 것이다.

위에서 기술한 다섯 가지 구문에 대한 힙 정보 변경에 관한 개념은 하위 절에서 기술한다.

3.1. new 연산자에 의한 새로운 객체할당

new 연산자를 이용해서 새로운 객체를 힙에 할당하면 참조변수는 새로 생성된 인스턴스를 참조한다. 그림 3은 new 연산자에 의해서 힙에 새로운 인스턴스가 만들어질 때 힙 사용 정보를 보여준다.

$Heap_{pre}$: []

```
OBJ ref = new OBJ();
```

$Heap_{post}$: [<OBJ, ref>]

그림 3. new 연산자에 의한 힙 상태 변경

그림 3에서 $Heap_{pre}$는 new 문장 수행 전의 힙 상태를 나타내고, $Heap_{post}$는 객체 할당이 수행된 후의 힙 상태를 보여준다. $Heap_{post}$는 노드로서 튜플 하나를 갖는다. 이 튜플의 첫 번째 원소는 생성된 인스턴스이고, 두 번째 원소는 인스턴스를 참조하는 참조변수명이다.

3.2. 배정문에 의한 객체 참조 변경

배정문은 참조변수가 다른 인스턴스를 참조하도록 변경할 수 있다. 이전에 참조되던 인스턴스가 다른 객체에 의해서 더 이상 참조되지 않는다면 이 인스턴스는 쓰레기 객체가 된다. 그림 4는 배정문에 의해서 변경된 힙 사용 정보를 보여준다.

$Heap_{pre}$: [<OBJ1, ref1>,<OBJ2,ref2>]

```
ref1 = ref2;
```

$Heap_{post}$: [<OBJ1,null>,<OBJ2,ref2,ref1>]

그림 4. 배정문에 의한 힙 상태 변경

배정문 수행 후 참조변수 ref1은 인스턴스 OBJ1의 튜플에서 제거되고 ref2가 참조하고 있는 OBJ2 튜플에 추가된다. OBJ에 부여된 번호는 동일한 클래스로부터 생성된 다른 인스턴스임을 의미한다. 그림 4의 <OBJ1, null> 튜플과 같이 참조 변수가 null인 경우는 이 객체가 쓰레기임을 의미한다.

3.3. 메서드 인수로서 참조 전달

메서드 호출 시 실인수로서 참조를 전달할 수 있다. 이 경우 메서드 내에서 이 인수를 배정 받는 참조변수는 현재 실인수가 참조하고 있는 인스턴스를 참조하게 된다. 그림 5는 메서드의 실인수로 참조변수가 전달되는 경우에 힙 사용 정보가 변경되는 것을 보여준다.

$Heap_{pre}$: [<OBJ1, ref1>,<OBJ2,ref2>]

```
m1(ref1);
```

$Heap_{post}$: [<OBJ1,ref1,ref2>,<OBJ2,null>]

그림 5. 메서드 인수에 의한 힙 상태 변경

그림 5에서 m1()은 인수로서 참조를 가지는 메서드다. 프로그램 실행 시에 m1()의 실인수로 참조 변수 ref1이 전달되고, m1()의 몸체에서 ref2의 참조는 ref1이 참조하는 객체로 변경되는 상황을 기술한 것이다. 그림 5에서 ref2는 클래스 소속 변수다. 만일 ref2가 지역 변수라면 메서드 m1() 수행이 끝난 후, ref2는 객체 튜플로부터 제거된다. 이 경우 힙 상태는 다음과 같이 변경된다.

$Heap_{post}$: [<obj1, ref1>, <obj2, null>]

3.4. 메서드 반환 값으로 참조 전달

참조를 반환하는 메서드가 배정문의 우변값이나 메서드 호출 시 실인수로 사용되는 경우이다. 그림 6은 참조를 반환하는 메서드에 의해서 변경된 힙 사용 정보를 보여준다.

$Heap_{pre}$: [<OBJ1,ref1>,<OBJ2,ref2>]

```
ref1 = m2();
```

$Heap_{post}$: [<OBJ1,null>,<OBJ2,ref2,ref1>]

그림 6. 메서드 반환 값에 의한 힙 상태 변경

그림 6의 경우 배정문의 경우와 유사하다. ref1은 ref2가 가리키던 객체로 참조가 변경된다. ref1은 반환문에 의해서 전달된 참조변수이다. 그림 6의 경우 ref2는 클래스 멤버 변수로 생각할 수 있다. 만일 ref2가 지역변수라면 3.3 절의 경우와 같이, OBJ2 객체의 튜플로부터 제거된다.

3.5. 블록 구조

블록 내부에서 선언된 참조변수들은 인스턴스를 참조할 수 있다. 블록 내부에서 선언된 참조변수들은 블록의 종료 지점에 이르면 인스턴스 튜플로부터 모두 제거되어야 한다. 그림 7은 블록에 의해서 변경된 힙 사용 정보를 보여준다.

$Heap_1$: [<OBJ1, ref1>,<OBJ2, ref2>]

```
{
    OBJ ref3 = new OBJ();
```

$Heap_2$: [<OBJ1, ref1>,<OBJ2, ref2>,<OBJ3, ref3>]

```
    ref1 = ref3
```

$Heap_3$: [<OBJ1, null>,<OBJ2, ref2>,<OBJ3, ref3, ref1>]

```
}
```

$Heap_4$: [<OBJ1, null>,<OBJ2, ref2>,<OBJ3,ref1>]

그림 7. 블록구조에 의한 힙 상태 변경

그림 7의 경우, ref3은 지역 참조 변수이다. 이 변수는 블록을 벗어나기 직전 ref3이 참조하는 튜플로부터 제거되고, 블록 외부에서 선언된 참조변수 ref1은 여전히 객체 OBJ3을 참조함을 의미한다.

3.6. 실행 예

이 절에서는 앞서 기술한 내용에 대한 몇 가지 예를

보이고자 한다. 힙에 할당된 객체는 쓰레기 판별을 위해서 인스턴스별로 유지된다. 수집된 정보에서 인스턴스는 "클래스명-번호"로 표현된다. 프로그램이 실행되는 동안 할당된 객체들의 정보는 클래스 수준에서 유지되고 관리된다. 블록 내에서 선언된 지역 참조변수들은 블록문을 벗어날 때 인스턴스 튜플로부터 제거되어야 하기 때문에 블록 단위로 별도의 지역 변수 리스트를 유지한다. 그림 8과 9 그리고 10은 본 논문에서 제시한 방법으로 계산한 힙 사용 정보이다. 이 그림들은 논문의 사정상 부분적으로 편집되었다.

- new 연산자에 의해서 할당된 객체의 힙 사용 정보 추출을 위한 코드 삽입

그림 8은 new 연산자에 의해서 변경된 힙 사용 정보를 추출하기 위해서 코드를 삽입하는 예이다.

```
<코드 삽입 전>
 date1 = new DateFirstTry();

<코드 삽입 후>
 date1 = new DateFirstTry();
 manOT.InsertAssign ("DateFirstTry_4",
                     "date_main_0");

<힙 상태 정보>
 DateFirstTry_4 : date1_main_0->null
```

그림 8. new 연산자에 의한 힙 정보 변경 처리

그림 8에서 manOT는 힙에 할당된 객체들의 정보를 저장하기 위한 리스트이고, InsertAssign()은 객체가 사용하는 힙 정보를 manOT에 추가하거나 변경하는 메서드이다. 저장된 힙 상태 정보는 참조변수들의 이름 충돌을 회피하기 위해서 "변수이름_선언된 메서드 이름_호출 수준"으로 표시된다. 호출 수준은 재귀호출 시 발생할 수 있는 이름 충돌을 회피하기 위해서 삽입되었다.

- 배정문에 의해서 변경된 객체의 힙 사용 정보 추출을 위한 코드 삽입

그림 9는 배정문이 참조변수가 참조하는 객체를 변경하는 경우, 변경된 힙 사용 정보를 추출하기 위한 코드가 삽입된 것이다. SearchRefObject()는 인수로 주어진 참조변수가 현재 참조하고 있는 객체의 이름을 결과로 반환하는 메서드다.

- 메서드 호출 시 참조에 의해서 변경된 힙 사용 정보 추출을 위한 코드 삽입

그림 10은 메서드 인수로 전달된 참조에 의해서 변경된 힙 사용 정보를 추출하기 위해서 코드를 삽입한 예이다. 지역 변수의 경우 블록을 벗어날 때 현재 참조하고 있는 객체의 리스트로부터 제거되어야 하기 때문에, 지역 참조변수 리스트에 추가된다. 그림 10에서는 vl_1.AddNode()가 이 일을 수행하고 있다.

```
<코드 삽입 전>
tdate = date2;

<코드 삽입 후>
tdate = date1;
manOT.InsertAssign(manOT.getObjTable().SearchRefObject
("date1_main_0"), "tdate_main_0");

<문장 실행 전 힙 상태>
DateFirstTry_4 : tdate_main_0->null
DateFirstTry_6 : date2_main_0->null

<문장 실행 후 힙 상태>
DateFirstTry_4 : null
DateFirstTry_6 : date2_main_0->tdate_main_0->null
```

그림 9. 배정문에 의한 힙 정보 변경 처리

```
<코드 삽입 전>
DateMethod(date1);

public DateFirstTry DateMethod(DateFirstTry df)
{
        DateFirstTry lvar;
        lvar = df;
        …
}

<코드 삽입 후>
vl_1.AddNode("lvar_DateMethod_1", "DateFirstTry");
lvar = df;
manOT.InsertAssign(manOT.getObjTable().SearchRefObject
("df_DateMethod_1"), "lvar_DateMethod_1");

<문장 실행 전 힙 상태>
DateFirstTry_4 : date1_main_0->null

<문장 실행 후 힙 상태>
DateFirstTry_6 : date1_main_0->df_DateMethod_1->
                 lvar_DateMethod_1->null
```

그림 10. 메서드 반환 결과에 의한 힙 정보 변경 처리

그림 10에서 lvar는 메서드 DateMethod() 내에서 선언되었기 때문에 DateMethod()의 실행이 종료될 때 date1의 튜플로부터 제거되어야 한다. 이를 위해서 lvar는 지역참조 변수 리스트에 추가된다. 배정문에 의해서 lvar가 참조할 객체는 df가 참조하고 있는 객체이다. 여기서 df는 직접적으로 어떤 객체를 참조하는 것이 아니라 참조변수 date1을 참조한다. 그렇기 때문에 lvar는 df가 아니라 date1이 참조하는 객체의 튜플에 추가되어야한다. 그림 10에서 SearchRefObject()는 간접 참조 변수가 참조하는 실제 객체를 반환한다.

4. 실험 및 분석

4절에서는 본 논문에서 제시된 방법을 실험한 결과에 대해서 기술한다. 실험은 공개되고 자주 사용되는 자바 응용 프로그램을 대상으로 하였다. 표 1은 실험에 사용한 프로그램에 대한 간략한 설명이다.

표 2는 본 실험에서 구현된 코드 삽입기를 이용해서

최대 힙 사용량을 분석한 결과이다. 본 실험에서 사용된 코드 삽입기의 경우, 모든 힙 사용 정보 변환 시점에서 쓰레기인 객체를 판별하기 때문에 일반적으로 사용되는 프로파일러와 힙 사용량을 비교 하는 것은 무의미하다. 표 5의 결과에서 힙 사용량은 미리 계산된 객체의 크기와 할당된 객체의 회수를 곱한 것이다.

표 1. 실험 프로그램의 간략한 설명

프로그램	간단한 설명	클래스 수	메서드 수	라인 수
ChangeDate	폴더의 날짜를 변경	1	9	85
JavaConverter	텍스트 파일을 다른 포맷으로 변경	1	5	189
Statistician	클래스의 메서드에 대한 간단한 통계	3	21	644
Zip	Zip 압축 및 복원	1	1	43

표 2. 최대 힙 사용량

프로그램	최대 할당된 객체 수	최대 힙 사용량(B)
ChangeDate	5	112
JavaConverter	9	226
Statistician	9	184
Zip	4	144

본 논문에서 제시하는 방법은 가상 기계를 변경하는 것이 아니라 소스 프로그램에 힙 사용 정보 추출을 위한 코드를 삽입하기 때문에 대상 프로그램의 크기를 증가시킨다. 그 결과로 실행 속도도 상당히 느려짐을 보였다. 그러나 실제 배포되는 프로그램에는 힙 사용 정보 추출을 위한 코드가 포함되지 않기 때문에 코드 크기와 실행 속도의 증가는 문제가 되지 않는다.

5. 결 론

본 연구에서는 코드 삽입 기법을 이용해서 힙 사용량을 프로파일링 하는 방법을 제안했다. 본 연구의 공헌은 다음과 같다.

- 자바에서 제공하는 프로파일러 인터페이스를 지원하지 않는 자바 가상기계에서도 힙 사용량을 분석할 수 있다.
- 특정 쓰레기 수집방식에 의존하지 않고 실행 중에 쓰레기를 판별하기 때문에 정확한 힙 사용 정보를 제공한다.
- 힙 사용량뿐만 아니라 각 실행지점에서객체의 사용 형태에 관한 정보를 제공한다.

본 연구의 단점은 다음과 같다.

- 힙에 표현되는 객체의 형태는 가상기계에 의존하기 때문에 힙 사용량을 정확한 수치로 표시하기 어렵다.
- 코드 삽입을 이용한 힙 사용량 프로파일링 기법은 소스코드를 기반으로 하기 때문에 소스코드가 없는 클래스 파일에 대해서는 적용할 수 없다는 한계점을 갖는다.

향후 과제로는 코드 삽입기의 성능을 개선을 위해서 정적 분석 기법과 선택적으로 힙 사용 정보를 출력하는 방법을 적용할 예정이다.

참고문헌

[1] http://en.wikipedia.org/wiki/Profiler
[2] http://java.sun.com/j2se/1.5.0/docs/guide/jvmti/
[3] http://java.sun.com/j2se/1.4.2/docs/guide/jvmpi
[4] http://java.sun.com/developer/technicalArticles/Programming/HPROF.html
[5] http://www.ej-technologies.com/products/jprofiler/overview.html
[6] http://jira.jboss.com/jira/browse/JBPROFILER
[7] http://www.netbeans.org/
[8] http://java.sun.com/docs/books/vmspec/2nd-edition/html/VMSpecTOC.doc.html