韓國 컴퓨터情報學會 論文誌
第12卷 第2號, 2007. 5.

2007-12-2-1-4

코드삽입을 이용한 자바프로그램의 힙 사용정보 분석기법

주 성 용*, 조 장 우**

A Technique to Analyze Heap Usage of Java Programs Using Code Inserting

Seongyong Joo *, Jang-Wu Jo **

요 약

자바에서는 가상기계와 프로파일러의 통신을 목적으로 JVM TI 같은 프로파일러 인터페이스를 제공한다. 그러나 자바 가상기계 구현명세는 프로파일러 인터페이스 구현을 요구하지 않는다. 따라서 프로파일러 인터페이스를 제공하지 않는 자바 가상기계에서는 JVM TI를 사용하는 프로파일러는 이용할 수 없다. 이러한 문제를 해결하기 위해서 코드 삽입 기법을 이용해서 프로파일러 인터페이스를 사용하지 않고 힙 사용정보를 분석하는 기법을 제안한다. 제안한 방법을 검증하기 위해서 코드 삽입기를 구현하였다. 실험은 공개되고 자주 사용되는 자바 응용 프로그램을 대상으로 하였고, 실험결과는 대상 프로그램에서 생성된 객체의 수와 최대 할당된 객체의 수 그리고 객체의 사용형태를 보여준다.

Abstract

In the Java a profiler interface such as JVM TI is provided for communication between a Java virtual machine and a profiler. The JVM specification, however, does not require an implementation of a profiler interface. Consequently the JVM without an implementation of a profiler interface cannot use a profiler with the JVM TI. For solving the problem, we propose a technique which analyzes heap usage without a profiler interface. This technique inserts codes to extract heap usage into source files in the Java. We implemented a code inserter to verify the proposed technique. We experimented with Java programs that are frequently used and opened, the experimental result presents the number of created objects, the maximum number of allocated objects, and the used pattern of the objects.

▶ Keyword : 프로파일링(Profiling), 코드 삽입(Code Insertion), 자바 가상기계(Java Virtual Machine), 동적분석(Dynamic Analysis), 힙 사용량(Heap Usage)

• 제1저자 : 주성용 • 교신저자 : 조장우
• 접수일 : 2007.3.9, 심사일 : 2007.4.11, 심사완료일 : 2007. 5.6.
* 동아대학교 컴퓨터공학과 박사과정, ** 동아대학교 컴퓨터공학과 부교수
※ 이 논문은 2004학년도 동아대학교 학술연구비(신진과제)에 의하여 연구되었음

I. 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 제공한다. 현재 자바 1.5버전에서는 프로파일러를 위한 표준 인터페이스로 JVM TI를 제공하며 이전 버전에서는 JVMPI를 제공한다[2][3]. JVM TI를 이용한 프로파일러로는 HProf과 JProfiler 그리고 JBossProfiler 등이 있고, NetBeans나 Eclipse 같은 통합개발환경에서도 JVM TI를 사용하는 프로파일러를 제공한다[4][5][6][7].

그러나 프로파일링을 위한 표준 인터페이스 구현은 자바 가상기계 명세의 요구사항이 아니다[8]. 그러므로 모든 자바 가상기계가 표준 프로파일러 인터페이스를 지원하는 것은 아니다. 특히 Java ME와 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 앞서 기술한 문제를 해결하기 위해서 표준 인터페이스를 사용하지 않는 힙 사용정보 분석 기법을 제안한다. 이 기법은 프로그램에서 사용하는 힙 사용정보를 분석하기 위해서 대상 프로그램의 소스 파일을 분석하고, 소스 파일이 힙 상태를 변경하는 구문을 포함한다면 힙 사용 정보를 추출하기 위한 코드를 삽입한다. 삽입된 코드를 포함하는 프로그램은 자바 가상기계에서 실행되면서 힙 사용 정보를 추출하고 보고한다. 이 기법에서는 정확한 힙 사용정보 분석을 위해서 쓰레기 객체를 고려한다.

Java ME에서 사용되는 자바 가상 기계인 KVM[9]이 대상으로 하는 시스템이나 임베디드 장치에서 메모리는 제한된 자원이며 데스크 탑 환경에서 사용되는 메모리 자원보다 빈약하다. 또한 이런 시스템들은 가상 메모리를 지원하지 않는 경우가 많다. 이런 시스템에서는 응용프로그램이 필요로 하는 메모리를 예측할 필요가 있다.

본 논문에서 제시하는 기법의 장점은 자바에서 제공되는 표준 프로파일러 인터페이스를 사용하지 않고 힙 사용정보를 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 방식이 가상기계 구현에 의존하기 때문이다.

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

II. 연구배경

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

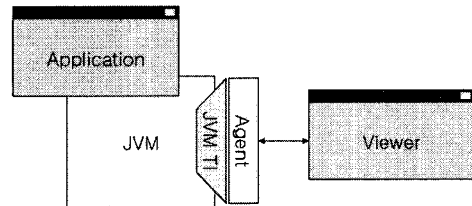

그림 1. JVM TI를 사용하는 프로파일러 동작방식
Fig 1. Profiler operation mode using the JVM TI

그림 1에서 Application은 힙 사용정보 분석을 위한 대상 프로그램이고, JVM은 자바 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. 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. 가상기계가 프로파일러 인터페이스를 지원하지 않는다면 이런 인터페이스를 이용하는 프로파일러를 사용할 수 없다. 이 같은 문제를 해결하기 위해서는 프로파일러 인터페이스를 사용하지 않는 프로파일링 기법이 요구된다.

힙 사용정보를 추출하는 코드를 소스 파일에 삽입함으로써 이 같은 문제를 해결한다. 제안한 방법은 대상 프로그램을 실행중인 가상기계가 사용하는 힙 정보를 추출하는 것이 아니라, 대상 프로그램의 소스 파일에 힙 사용 정보를 변경하는 구문이 존재한다면 그 구문에 힙 사용 정보 추출을 위

한 코드를 삽입한다. 코드가 삽입된 프로그램은 자바 가상 기계에서 실행되면서 힙 사용 정보를 추출한다. 그림 2는 코드 삽입을 이용한 힙 사용정보 분석 기법의 동작방식이다.

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.

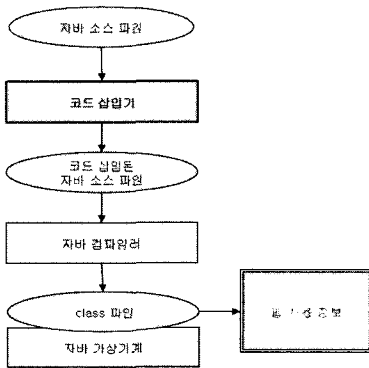

그림 2. 코드 삽입기를 이용한 힙 정보 추출 동작 방식
Fig 2. Extracting heap usage using the code insertor

제안한 방식과 같이 프로그램을 실행하면서 동적인 속성들을 평가하는 기존의 연구로 [10,11,12,13,14,15]가 있다. [10]에서는 실행시간 모니터링 기법을 이용해서 대상 응용프로그램의 쓰레드 동기화에 대해서 연구하였고, [11, 12, 13, 14, 15] 에서는 응용프로그램 실행 시 지정된 보안정책에 순응하는지를 평가하기 위해서 참조 모니터를 이용하는 방법을 제안했다. 제안한 방법이 실행 시간에 참조 모니터를 이용하여 프로그램을 분석하고 평가한다는 점에서는 기존연구와 유사하나, 기존 연구들의 대상이 쓰레드와 보안정책인 반면 본 연구는 힙을 대상으로 한다는 점에서 다르다.

III. 코드삽입을 이용한 힙 사용정보 분석

힙 사용정보는 힙 사용량과 힙에 할당된 객체를 참조하는 변수를 분석하는 것이다. 힙 사용량 계산을 위한 가장 단순한 방법은 new 연산자에 의해 할당된 객체를 분석하는 것이다. 그러나 이 방법은 쓰레기를 고려하지 않기 때문에 결과는 쓰레기 객체로 회수된 힙 사용량을 포함하는 결과를 보여주게 된다. 정확한 힙 사용량을 계산하기 위해서는 쓰레기 판별이 요구된다. 본 논문에서는 쓰레기 판별을 위해서 각 객체들을 참조하는 변수들의 정보를 유지한다. 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 참조를 반환하는 메서드
⑤ 블록 구조

위 다섯 가지 구문을 설명하기 위해서 힙을 객체의 이름과 그 객체에 대한 참조 변수들의 튜플로 구성된 리스트로 표현한다. 예를 들면 다음과 같다.

· 힙에 객체 OBJ가 할당되어 있고, 이 객체를 참조 변수 var1과 var2가 참조하는 경우

```
Heap:[<OBJ, var1, var2>]
```

· 힙에 OBJ1과 OBJ2가 할당되어 있고, 이 OBJ1을 참조 변수 var1이 참조하고 OBJ2를 var2가 참조하는 경우

```
Heap:[< OBJ1, var1>, <OBJ2, var2>]
```

· 힙 사용 정보가 변경되는 경우, 변경전의 상태는 $Heap_{pre}$, 변경후의 상태는 $Heap_{post}$로 표현한다.

```
Heap_pre:[<OBJ1, var1>, <OBJ2, var2>]
Heap_post:[<OBJ1, null>,
        <OBJ2, var2, var1>]
```

마지막 단락은 $Heap_{pre}$ 상태에서는 변수 var1이 OBJ1을 참조하다가 $Heap_{post}$에서 OBJ2로 참조가 변경되는 상태를 보인 것이다.

3.1 new 연산자에 의한 새로운 객체할당

new 연산자를 이용해서 새로운 객체를 힙에 할당하면 참조 변수는 새로 생성된 객체를 참조한다. 그림 3은 new 연산자에 의해서 힙에 새로운 객체가 만들어질 때 힙 사용 정보를 보여준다.

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 원소로서 튜플 하나를 갖는다. 이 튜플의 첫 번째 원소는 생성된 객체이고, 두 번째 원소는 객체를 참조하는 참조 변수명이다.

```
Heap_pre: []
OBJ ref = new OBJ();
Heap_post: [<OBJ, ref>]
```

그림 3. new 연산자에 의한 힙 상태 변경
Fig 3. Heap usage changed by the new operator

3.2 배정문에 의한 객체 참조 변경

```
Heap_pre: [<OBJ1, ref1>,<OBJ2,ref2>]
ref1 = ref2;
Heap_post: [<OBJ1, null>,<OBJ2,ref2,ref1>]
```

그림 4. 배정문에 의한 힙 상태 변경
Fig 4. Heap usage changed by an assignment

배정문은 참조 변수가 다른 객체를 참조하도록 변경할 수 있다. 이전에 참조되던 객체가 더 이상 참조되지 않는다면 이 객체는 쓰레기가 된다. 그림 4는 배정문에 의해서 변경된 힙 사용 정보를 보여준다.

배정문 수행 후 참조 변수 ref1은 객체 OBJ1의 튜플에서 제거되고 ref2가 참조하고 있는 OBJ2 튜플에 추가된다. OBJ에 부여된 번호는 동일한 타입의 다른 객체임을 의미한다. 그림 4에서 〈OBJ1, null〉은 이제 쓰레기임을 보여준다.

3.3 메서드 인수로서 참조 전달

메서드 호출 시 실인수로서 참조를 전달하면 메서드 내부에서 인수를 배정받는 참조 변수는 실인수가 참조하는 객체로 참조가 변경된다. 그림 5는 메서드의 실인수로 참조 변수가 전달되는 경우에 힙 사용 정보가 변경되는 것을 보여준다.

그림 5에서 m1()은 메서드다. 그림 5에서 ref2는 객체의 멤버 변수이고, m1()의 몸체에서 ref2는 ref1이 가리키는 객체로 참조가 변경된다.

```
Heap_pre: [<OBJ1,ref1>,<OBJ2,ref2>]
m1(ref1);
Heap_post: [<OBJ1,ref1,ref2>,<OBJ2,null>]
```

그림 5. 메소드 인수에 의한 힙 상태 변경
Fig 5. Heap usage changed by a parameter of method

3.4 참조를 반환하는 메서드

그림 6은 참조를 반환하는 메소드에 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
Fig 6. Heap usage changed by a return value of method

그림 6은 배정문의 경우와 유사하다. m2()는 ref2의 참조를 반환하는 메서드이고, ref2는 객체 멤버 변수이다. 이 경우 ref1은 ref2가 가리키는 객체로 참조가 변경된다. 만일 m2()가 반환하는 것이 새로운 객체라면 $Heap_{post}$는 다음과 같이 변경된다.

```
Heap_post: [<OBJ1,null>,<OBJ2,ref2>,
            <OBJ3,ref1>]
```

3.5 블록 구조

블록 내부에서 선언된 참조 변수들은 객체를 참조할 수 있다. 블록 내부에서 선언된 지역 참조 변수들은 블록의 종료 지점에 이르면 객체의 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
Fig 7. Heap usage changed by a block structure

그림 7의 경우, ref3은 지역 참조 변수이다. 그러므로 ref3은 블록을 벗어나기 직전 OBJ3 튜플로부터 제거되고, 블록 외부에서 선언된 참조 변수 ref1은 여전히 객체 OBJ3을 참조한다.

3.6 코드 삽입 예

이 절에서는 앞서 기술한 내용에 대한 몇 가지 예를 보이고자 한다. 객체를 참조하는 변수들의 정보는 쓰레기 객체 판별을 위해서 객체 별로 유지된다. 수집된 정보에서 객체는 '타입명_번호'로 표현된다. 대상 프로그램에서 할당되는 객체들의 정보는 클래스 수준에서 관리된다. 지역 변수는 블록 문을 벗어날 때 객체를 참조하는 변수들의 리스트에서 제거되어야 하기 때문에, 각 블록 별로 지역 변수 리스트를 유지한다.

• new 연산자를 이용한 새로운 객체 할당을 위한 코드 삽입

그림 8은 new 연산자에 의해서 힙에 할당된 객체의 정보를 추출하기 위한 코드를 삽입하는 예이다.

그림 8에서 manOT는 힙에 할당된 객체들의 정보를 관리하기 위한 테이블이다. InsertAssign()은 객체를 참조하는 변수에 대한 정보를 추가하거나 변경하는 메서드다. 참조 변수는 저장 시 이름 충돌을 회피하기 위해서"변수이름_선언된-메서드명_호출수준"으로 표시된다. 호출수준은 재귀 호출 시 발생할 수 있는 이름 충돌을 회피하기 위해서 삽입되었다.

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

<코드 삽입 후>
date1 = new DateFirstTry();
manOT.InsertAssign ("DateFirstTry_4",
                    "date1_main_0");

<문장 실행 후 힙 상태>
DateFirstTry_4 : date1_main_0->null
```

그림 8. new 연산자에 의한 힙 정보변경 처리
Fig 8. Processing for heap usage changed by the new operator

• 배정문에 의한 객체 참조 변경을 위한 코드 삽입

그림 9는 배정문에 의해서 참조 변수의 참조가 변경된 경우다. SearchRefObject()는 인수로 주어진 참조 변수가 현재 참조하고 있는 객체의 이름을 반환하는 메서드이다.

```
<코드 삽입 전>
tdate = date1

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

<문장 실행 전 힙 상태>
DateFirstTry_4 : tdate_main_0->null
DateFirstTry_6 : date1_main_0->null

<문장 실행 후 힙 상태>
DateFirstTry_4 : null
DateFirstTry_6:
      date1_main_0->tdate_main_0->null
```

그림 9. 배정문에 의한 힙 정보변경 처리
Fig 9. Processing for heap usage changed by an assignment

• 메서드 호출 시 인수로써 참조 변수 전달을 위한 코드 삽입

```
<코드 삽입 전>
DateMethod(date1);

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

<코드 삽입 후>
v1_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_4 :
date1_main_0->df_DateMethod_1->
              lvar_DateMethod_1->null
```

그림 10. 메소드 반환 결과에 의한 힙 정보변경 처리
Fig 10. Processing for heap usage changed by a return value of method

표 1. 대상 프로그램의 간략한 설명
Table 1. List of benchmarks and application used in the evaluation

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

그림 10은 전달된 인수에 의해서 객체 참조가 변경되는 경우를 처리하는 코드 삽입 예이다. 지역 변수의 경우 블록을 벗어날 때 현재 참조하고 있는 객체의 리스트로부터 제거되어야 하기 때문에, 지역 참조 변수 리스트에 추가된다. 그림 10에서 vl_1.AddNode()가 이 일을 수행하고 있다.

메서드 DateMethod()의 가인수 df는 실제로 지역변수이기 때문에 lvar에 대한 배정문에서 lvar가 실제로 참조해야 할 객체는 date1이 참조하는 객체다. 이 문제를 해결하기 위해서 메서드 호출 전 가인수 df를 실인수 date1이 참조하는 객체를 참조하도록 변경한다. 나머지 경우도 유사한 방법으로 해결하였다.

IV. 실험 및 분석

본 논문에서는 실험을 위해서 본 논문에서 제안한 코드 삽입기를 구현하였다. 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 표 1의 응용프로그램들의 힙 사용량을 분석한 결과이다. 표 2에서 '생성된객체수'는 각 응용프로그램에서 new 연산으로 생성된 객체의 개수이고, '최대할당된객체수'는 생성된 객체들에서 쓰레기 객체를 제외한 객체 수에서 가장 많을 때의 개수이다.

그림 11은 코드 삽입기를 이용해서 프로그램 JavaConverter의 힙 사용정보를 분석한 결과이다. 그림 11의 결과는 객체의 할당량과 특정 지점에서 객체가 참조되는 방식과 참조가 변경되는 형태와 언제 쓰레기가 되는지를 명시적으로 보여주며, 각 실행지점에서 객체의 사용형태에 관한 정보를 제공한다.

제안한 방법은 힙 사용정보를 분석하기 위한 코드가 대상 응용프로그램에 포함되기 때문에 기존의 인터페이스를 이용한 방법보다 잠재적으로 더 많은 정보를 분석할 수 있음을 보여 준다.

그러나 힙에 표현되는 객체의 형태는 가상기계에 의존하기 때문에 힙 사용량을 정확한 수치로 표현하기 어렵다. 또한 코드 삽입을 이용한 힙 사용량 프로파일링 기법은 소스코드를 기반으로 하기 때문에 소스코드가 없는 클래스 파일에 대해서는 적용할 수 없다.

표 2. 코드 삽입기로 분석한 할당된 객체수와 최대 힙 사용량
Table2. Numbers of Objects Allocated and Amounts of Used Heap Analyzed by the Code Insertor

프로그램	생성된객체수	최대할당된객체수
ChangeDate	7	5
JavaConverter	9	9
Statistician	11	9
Zip	5	5

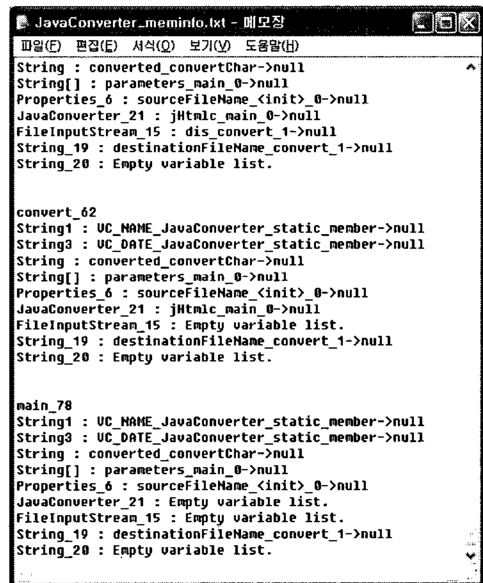

그림 11. 본 논문에서 제시한 힙 사용정보 분석화면
Fig 11. Picture of Heap Usage Analyzed by the Proposing Technique in This Paper

본 논문에서는 제안한 방식과 기존 JVM TI를 사용하는 프로파일러와의 비교를 포함하지 않았다. 그 이유는 제안한 방식과 프로파일러는 분석 대상이 다르며, 제안한 방식은 기존 JVM TI를 사용하는 프로파일러가 동작할 수 없는 환경을 대상으로 하였기 때문이다.

V. 결론 및 향후과제

본 논문에서는 표준 프로파일러 인터페이스를 제공하지 않는 자바 가상기계에서 힙 사용량을 평가하기 위한 방법을 제안했다. 제안된 방법은 자바 소스 코드에 힙 사용정보를 변경하는 구문이 포함된다면 힙 사용정보를 추출하기 위한 코드를 삽입한다. 힙 사용정보 추출코드를 포함한 자바 소스 프로그램은 컴파일 되고 실행하면서 힙 사용정보를 반환한다. 실험에서는 힙 사용정보 변경 구문을 식별하고 힙 사용정보를 추출하기 위해서 코드 삽입기를 구현하였다.

휴대전화나 임베디드 장치와 같이 소용량 메모리를 사용하는 장치들은 메모리 제약으로 인해서 JVM TI와 같은 표준 프로파일러 인터페이스를 지원하지 않을 수 있다. 제안된 방법은 자바 가상기계를 수정할 필요가 없이 힙 사용정보를 추출할 수 있기 때문에, JVM TI와 같은 프로파일러 인터페이스를 지원하지 않는 장치를 대상으로 하는 자바 응용프로그램들의 힙 사용정보를 분석하기 위해서 제안한 방법은 유용할 것으로 사료된다.

또한 제안한 방법은 힙 사용정보를 분석하기 위한 코드가 대상 응용프로그램에 포함되기 때문에 기존의 인터페이스를 이용한 방법보다 잠재적으로 더 많은 정보를 분석할 수 있다.

향후 과제로 힙 사용량뿐만 아니라 프로그램의 다른 동적 속성들을 평가하도록 코드 삽입기를 확장하고, 코드 삽입기의 성능 개선을 위해서 정적 분석 기법과 선택적으로 힙 사용 정보를 출력하는 방법을 적용할 예정이다.

참고문헌

[1] 프로파일러 정의, http://en.wikipedia.org/wiki/Profiler

[2] JVM TI Homepage, http://java.sun.com/j2se/1.5.0/docs/guide/jvmti/

[3] JVMPI Homepage, http://java.sun.com/j2se/1.4.2/docs/guide/jvmpi/

[4] HProf Homepage, http://java.sun.com/developer/technicalArticles/Programming/HPROF.html

[5] jprofiler Homepage, http://www.ej-technologies.com/products/jprofiler/overview.html

[6] JBProfiler Homepage, http://jira.jboss.com/jira/browse/JBPROFILER

[7] NetBeans Homepage, http://www.netbeans.org/

[8] JVMSpecification, http://java.sun.com/docs/books/vmspec/2nd-edition/html/VMSpecTOC.doc.html

[9] CLDC Homepage, http://java.sun.com/products/cldc

[10] Sewon Moon, Byeong-Mo Chang, "A Thread monitoring System for Multithreaded Java Programs," ACM SIGPLAN Notices Vol 41(5), May 2006.

[11] U. Erlingsson, The inlined reference Monitor approach to secure policy enforcement, Ph.D thesis, Cornell University, January 2004.

[12] Lujo Bauer, Jarred Ligatti, David Walker, "Types and Effects for Non_Interfering Program Monitors," International Symposium on Software Security. Tokyo, November, 2002.

[13] Ulfar Erlingsson , Fred B. Schneider, "SASI enforcement of security policies: A retrospective," Proceedings of the New Security Paradigms Workshop, Pages 87-95, September 1999.

[14] Fred B. Schneider, "Enforceable security policies," ACM Transactions on Information and System Security 3, 1, Pages 30-50, February 2000.

[15] Lujo Bauer, Jarred Ligatti, David Walker, "More Enforceable Security Policies," Workshop on Computer Security, July 2002.

저 자 소 개

주 성 용

2000년 동아대학교 컴퓨터공학과(학사)
2002년 동아대학교 컴퓨터 공학과 컴퓨터공학 전공(석사)
2002년 ~현재 동아대학교 컴퓨터공학과 컴퓨터공학 전공 박사 과정
관심분야 : 프로그래밍 언어, 컴파일러, 임베디드 시스템

조 장 우

1992 서울대학교 계산통계학과(학사)
1994 서울대학교 전산과학과 (석사)
2003 한국과학기술원 전산학과 (박사)
현재 동아대학교 컴퓨터공학과 부교수
관심분야: 프로그램 분석, 임베디드 시스템