Java 언어를 위한 쓰레드 모니터링 시스템

문 세 원† · 창 병 모††

요 약

본 연구에서는 견고한 멀티 쓰레드 소프트웨어 개발을 돕기 위한 쓰레드 모니터링 시스템을 개발하였다. 이 시스템은 쓰레드 실행 과정과 동기화 과정를 시각적으로 추적, 모니터링 할 수 있다. 또한 사용자는 실행 전에 옵션 선택을 통해 관심 있는 쓰레드와 동기화만을 선택하여 이들을 중점적으로 모니터링 할 수 있으며 실행 후에는 실행된 쓰레드와 동기화의 특징을 요약한 프로파일 정보를 제공한다. 본 연구에서는 코드 인라인 기법을 기반으로 이 시스템을 구현하였으며 실험을 통한 실험 결과를 통해 그 효율성을 보인다.

키워드 : 자바, 쓰레드, 모니터링

A Thread Monitoring System for Java

Sewon Moon† · Byeong-Mo Chang††

ABSTRACT

To assist developing robust multithreaded software, we develop a thread monitoring system for multithreaded Java programs, which can trace or monitor running threads and synchronization. We design a monitoring system which has options to select interesting threads and synchronizations. Using this tool, programmers can monitor only interesting threads and synchronization in more details by selecting options. It also provides profile information after execution, which summarizes behavior of running threads and synchronizations during execution. We implement the system based on code inlining, and presents some experimental results.

Key Words : Java, Thread, Monitoring

1. 서 론

현대 소프트웨어 개발에 있어서 멀티 쓰레드는 널리 이용되고 있으며 서버 소프트웨어처럼 병행적으로 반응하는 소프트웨어를 개발하는데 보편적으로 사용되고 있다[8]. 단일 쓰레드 프로그램에서는 한 순간에 한 쓰레드의 하나의 인스턴스만 실행되는 반면에 멀티 쓰레드 프로그램에서는 여러 개의 쓰레드가 병행적으로 실행될 수 있기 때문에 프로그램의 특징을 이해하기 쉽지 않으며 여러 개의 실행 쓰레드는 자원을 공유하기 때문에 실행하는 동안 레이스 컨디션이나 데드락 등이 발생할 수 있어 프로그램의 이해를 더욱 어렵게 한다.

따라서 견고한 멀티 쓰레드 프로그램을 개발하기 위해서는 프로그래머는 효과적으로 멀티 쓰레드 프로그램의 실행 과정을 관찰하는 것이 필요하나 여러 개의 실행 쓰레드와 동기화 과정을 효과적으로 추적하거나 관찰하는 것은 어려운 일이다. 따라서 견고한 멀티 쓰레드 소프트웨어 개발을 돕기 위해서 실행시간 동안 쓰레드 실행과 동기화 과정을 추적하거나 모니터링을 하는 도구가 필요하다.

본 연구에서는 멀티 쓰레드 자바 프로그램을 위한 동적 쓰레드 모니터링 시스템을 개발하였다. 이 시스템은 프로그램 실행 동안에 쓰레드 실행 과정과 동기화 과정을 시각적으로 모니터링 할 수 있도록 해준다. 이 도구를 이용하여 프로그래머는 실행 전에 옵션 선택을 통해 관심 있는 쓰레드와 동기화만을 선택하여 보다 자세하게 시각적으로 모니터링 할 수 있다. 이 시스템은 또한 실행 후에 실행 쓰레드와 동기화의 특징을 요약한 프로파일 정보를 제공한다. 본 연구는 성능 오버헤드를 줄이기 위해서 코드 인라인 기법[1]을 기반으로 쓰레드 모니터링 시스템을 구현하였다. 이 시스템은 자바 컴파일러 전단부인 Barat[3]을 이용하여 구현되었으며 이 시스템의 효용성을 보여줄 수 있는 실험 및 실험 결과를 제시한다.

본 논문은 다음과 같이 구성된다. 다음 절에는 동기를 부여하는 예제를 보여준다. 3절에서는 전반적인 시스템을 설명하고 4절에서는 시스템 구현에 관해 설명한다. 5절에서는 몇 가지 실험 결과를 보여주고 6절에서는 이 논문의 결과와 향후 연구에 대해서 논한다.

† 정 회 원 : 삼성전자 연구원
†† 정 회 원 : 숙명여자대학교 컴퓨터과학과 교수(교신저자)
논문접수 : 2006년 1월 19일, 심사완료 : 2006년 5월 23일

```
class Deallock3thread extends Thread{
    static Object a = new Object(); // a monitor object
    static Object b = new Object(); // b monitor object
    static Object c = new Object(); // b monitor object
    String name;

    public Deallock3thread(String name){
        this.name = name;
    }

    public void run() {
        if(name.equals("d1") ){
            synchronized(a) {
                System.out.println("d1 acquired a");
                try { Thread.sleep(1000); }
                catch (InterruptedException e) {}

                synchronized(b) {
                    System.out.println("d1 acquired b");
                }
            }
        }
        else if(name.equals("d2") ){
            synchronized(b) {
                System.out.println("d2 acquired b");
                try { Thread.sleep(1000); }
                catch (InterruptedException e) {}

                synchronized(c) {
                    System.out.println("d2 acquired c");
                }
            }
        }
        else if(name.equals("d3") ){
            synchronized(c) {
                System.out.println("d3 acquired c");
                try { Thread.sleep(1000); }
                catch (InterruptedException e) {}

                synchronized(a) {
                    System.out.println("d3 acquired a");
                }
            }
        }

    }
}
public class Deadlock3main {
    public static void main(String[] args) {
        Deallock3thread d1 = new Deallock3thread("d1");
        Deallock3thread d2 = new Deallock3thread("d2");
        Deallock3thread d3 = new Deallock3thread("d3");

        d1.start();
        d2.start();
        d3.start();
    }
}
```

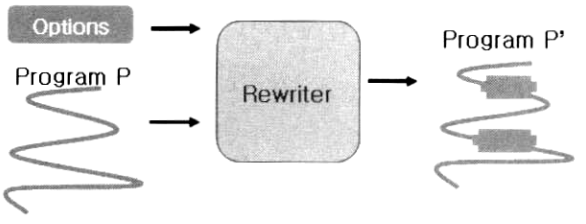
(그림 1) 예제 프로그램

2. 예 제

이 논문에서는 실행 예제로 (그림 1)의 멀티 쓰레드 자바 프로그램을 고려한다. 이 프로그램은 3개의 쓰레드를 생성하고, 각 쓰레드는 하나의 객체를 동기화한다. 그 후 다른 쓰레드에 의해 동기화된 다른 객체를 동기화하려고 시도한다. 이것은 실행 중에 데드락 상태를 만들 것이다.

3. 설계 고려사항

프로그램 실행을 모니터링 하는 방법은 3가지가 있다. 전통적인 레퍼런스 모니터는 기계 명령어 실행 전에 명령어와 함께 입력으로 받은 레퍼런스 모니터를 호출함으로써 구현된다. 두 번째 방법은 하드웨어 지원 없이 코드를 실행하고 각 명령어 전에 레퍼런스 모니터를 호출하는 방법으로 JVM과 같은 인터프리터 내에서 응용프로그램을 실행하는 것이다. 그러나 이 방법은 매 실행되는 명령어마다 비용이 발생하기 때문에 너무 큰 성능 오버헤드를 가지고 있다[3]. 세 번째 방법은 목적 소프트웨어 내에 레퍼런스 모니터를 삽입하는 방법으로 전통적인 레퍼런스 모니터의 제약점을 극복하면서 적절한 성능을 보여준다[3].



(그림 2) 인라인 모니터링

인라인 레퍼런스 모니터(IRM : Inlined Reference Monitor)는 레퍼런스 모니터의 기능을 포함하도록 어플리케이션 내에 삽입됨으로써 얻을 수 있다. (그림 2)처럼 IRM은 변환기(rewriter)에 의해 어플리케이션 내에 삽입된다. 이것은 어플리케이션과 옵션 혹은 정책 등을 읽고 해당 모니터링 코드가 삽입된 어플리케이션을 생성한다[3]. 변환된 프로그램을 실행하면 삽입된 코드에 의해서 실행 과정이 모니터링 된다.

본 연구에서는 쓰레드 모니터링 시스템을 설계하는데 있어서 시스템의 효율성과 효용성을 위해 다음 사항들을 고려하였다.

(1) **시각화** : 쓰레드 실행 과정 및 동기화 진행 과정을 시각적으로 보여줌으로써 프로그램을 효과적으로 모니터링 할 수 있도록 한다.

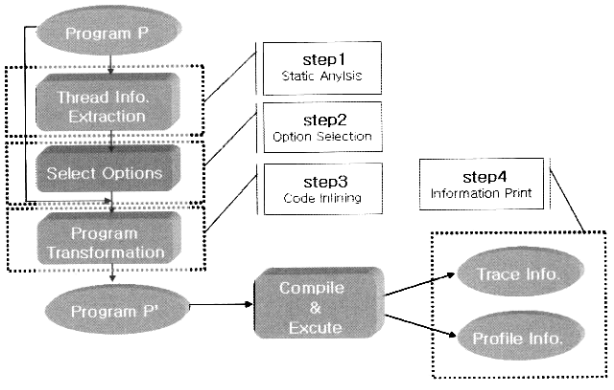
(2) **옵션 선택** : 관심 있는 쓰레드와 동기화만을 선택하여 모니터링 할 수 있도록 사용자에게 옵션을 제공한다. 이 기능은 모든 쓰레드나 동기화 대신 관심있는 대상만을 모니터링 하도록 일부 코드만 삽입하기 때문에 성능 오버헤드를 줄이는 데에도 기여할 수 있다.

(3) **프로파일 정보** : 실행 후에 쓰레드 실행과 동기화 과정 등을 요약한 프로파일 정보를 제공한다.

(4) **코드 인라인** : 전체 시스템의 효율성을 위해 성능 오버헤드를 줄이기 위한 코드 인라인 기법을 사용하였다.

입력 프로그램 P는 사용자 옵션에 따라서 관심 있는 쓰레드만을 추적할 수 있도록 코드를 삽입하여 프로그램 P′으로 변환한다. 변환된 P′은 실행시간 동안 쓰레드와 동기화 과정이 어떻게 시작되어 진행되는지 추적할 것이다. 그리고 실행 후에 쓰레드와 동기화에 관련된 요약 정보인 프로파일 정보를 제공한다.

전반적인 시스템 구조는 (그림 3)과 같다. 이 시스템은 (그림 3)에서 보이는 것처럼 4단계로 구성되어 있다. 각 단계별 기능은 다음과 같다:



(그림 3) 시스템 구조

(1) **정적 분석(Static Analysis) 단계** : 입력 프로그램을 정적 분석하여 쓰레드와 동기화 관련된 구문구조들을 추출한다. 구체적으로 쓰레드를 정의한 클래스와 관련 메쏘드들, 그리고 동기화 블록 등을 추출한다. 이 정적 분석 정보는 옵션 제공을 위해 이용된다.

(2) **옵션 선택(Option Selection) 단계** : 쓰레드와 동기화 관련된 정적 분석 정보를 이용하여 사용자에게 관심있는 쓰레드와 동기화 등을 선택하도록 한다. 이렇게 함으로써 실제 실행 시에 관심 있는 쓰레드와 동기화만을 모니터링 할 수 있다.

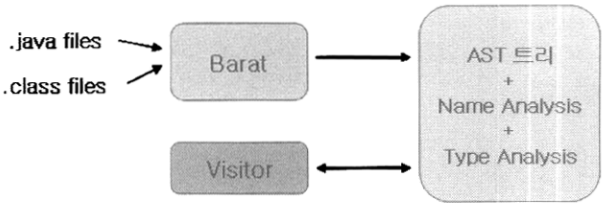
(3) **프로그램 변환(Program Transformation) 단계** : 사용자의 선택에 따라 해당 쓰레드와 동기화만을 모니터링하는 코드를 삽입하여 입력 프로그램 P를 새로운 프로그램 P′으로 변환한다.

(4) **프로그램 실행(Program Execution) 단계** : 변환된 프로그램 P′을 컴파일 하여 실행하는 단계이다. 이것은 프로그램에 따라 Java 2 SDK 또는 J2ME WTK에서 실행된다.

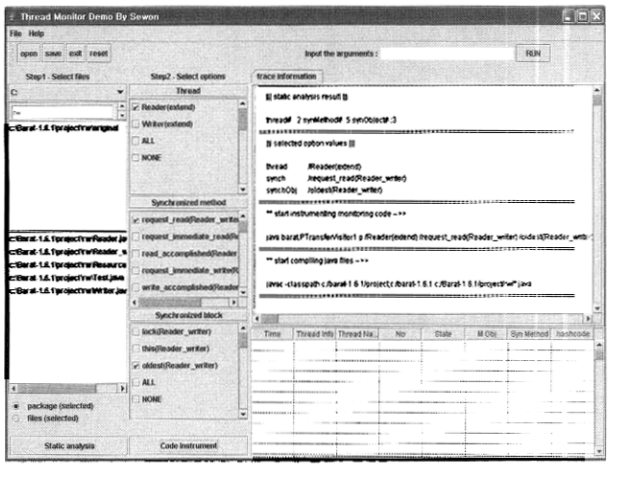
4. 구 현

4.1 구현 도구

본 연구에서는 정적 분석 및 프로그램 변환을 구현하기 위해 자바 컴파일러의 전단부인 Barat을 사용하였다. Barat은 입력으로 받은 자바 프로그램을 위해 추상 구문 트리(Abstract Syntax Tree)를 만들고 타입과 이름 분석 정보를 포함하여 이를 확장한다. 또한 비지터 디자인 패턴을 기반으로 추상 구문 트리를 선회하기 위한 인터페이스를 제공한다. 개발자는 추상 구문 트리 노드를 선회할 수 있는 트리 선회 루틴인 비지터를 확장하여 각 노드를 방문하면서 필요한 행동이나 연산을 할 수 있다. Barat은 기본적인 몇 개의 비지터를 제공하는데 DescendingVisitor는 각 추상 구문 트리의 노드를 깊이 우선 탐색 방법으로 선회한다. OutputVisitor는 추상 구문 트리의 노드를 선회하면서 입력 프로그램을 다시 프린트해준다. 본 연구에서는 추상 구문 트리의 노드를 방문할 때 필요한 행동이나 연산을 할 수 있도록 기본 비지터를 확장하여 정적 분석기를 구현한다.



(그림 4) Barat의 구조



(그림 5) 메인 화면

4.2 시스템 구현

본 시스템은 (그림 3)에서처럼 네 단계로 구성한다. 본 연구에서는 처음 세 단계를 구현한다. 마지막 단계는 Java SDK에서 실행한다. 이 시스템의 메인 화면은 (그림 5)와 같다. 샘플 입력 프로그램의 패키지 내 모든 파일이 리스트되어 있다. 사용자가 패키지에서 하나의 파일을 선택하면 창에는 정적 분석 정보를 기반으로 쓰레드, 동기화 함수, 동기화 블록 등의 리스트를 보여준다. 사용자는 이들 중 관심있는 쓰레드, 동기화 함수 등을 선택함으로써 관심 있는 대상만을 집중적으로 모니터링 할 수 있다.

본 연구에서는 입력 프로그램을 정적 분석하여 쓰레드와 동기화에 관련한 구문 구조들을 추출하고 이 분석 정보를 이용하여 사용자에게 관심 있는 쓰레드, 동기화 등을 선택할 수 있는 정보를 제공한다. 정적 분석기는 Barat이 제공하는 DescendingVisitor를 확장하여 AST 노드들을 선회하면서 쓰레드와 동기화에 관련한 구문구조를 방문할 때마다 해당 구문구조들만을 추출하도록 구현하였다.

정적 분석 정보를 이용하여 사용자가 관심 있는 쓰레드와 동기화 함수 등을 선택하면 입력 프로그램 P를 선택한 옵션 선택 정보를 이용하여 실행 쓰레드와 동기화 등을 모니터링 할 수 있는 코드를 삽입하여 새로운 프로그램 P′으로 변환한다. 본 연구에서는 이 프로그램 변환기를 OuputVisitor를 확장하여 구현하였다. (그림 6)은 이 프로그램 변환기의 전체적인 구조를 보여준다.

이 프로그램 변환기는 입력 프로그램의 AST 노드를 방문하면서 쓰레드, 동기화 함수, 동기화 블록 등을 만날 때

```
class PTransferVisitor extends OutputVisitor{

        public void visitConcreteMethod(ConcreteMethod o){
          // recode information about starting thread
        }
        public void visitConstructor(Constructor o){
          // recode thread name and type
        }
        public void visitObjectAllocation(ObjectAllocation o){
          // recode information about creating thread
        }
        public void visitInstanceMethodCall(InstanceMethodCall o){
          // recode information about calling synchronized method
        }
        public void visitBlock(Block o){
          //recode information about excuting synchronized method
        }
        public void visitSynchronized(Synchronized o){
          // recode information about object accessing synchronized object
        }
}
```

(그림 6) TransformVisitor 구조

```
public void run() {
  profile.TreatThread.manageThread2(Thread.currentThread().getName(), "ts" );
  try{
  if (this.name.equals("d1")) {
    try{
      profile.TreatThread.deadlockCheck1("a(Deallock3thread)", "a", Thread.currentThread().getName() );
      profile.TreatThread.nearSObj2("Deallock3thread", "a", Thread.currentThread().getName() );
      profile.TreatThread.manageThread2(Thread.currentThread().getName(), "ea" );

    synchronized (a) {
        profile.TreatThread.deadlockCheck2("a(Deallock3thread)", Thread.currentThread().getName());
        profile.TreatThread.startSObj2("Deallock3thread", "a", Thread.currentThread().getName() );
        profile.TreatThread.manageThread2(Thread.currentThread().getName(), "os" );
        java.lang.System.out.println("d1 acquired a");

        try { Thread.sleep(4000);}
        catch (InterruptedException e) {}
        finally { }
        try{
        profile.TreatThread.deadlockCheck1("b(Deallock3thread)", "b", Thread.currentThread().getName() );
        profile.TreatThread.nearSObj2("Deallock3thread", "b", Thread.currentThread().getName() );
        profile.TreatThread.manageThread2(Thread.currentThread().getName(), "ea" );

          synchronized (b) {
              profile.TreatThread.deadlockCheck2("b(Deallock3thread)", Thread.currentThread().getName());
              profile.TreatThread.startSObj2("Deallock3thread", "b", Thread.currentThread().getName() );
              profile.TreatThread.manageThread2(Thread.currentThread().getName(), "os" );
              java.lang.System.out.println("d1 acquired b");
          }
        }finally{
          profile.TreatThread.stopSObj2("Deallock3thread", "b", Thread.currentThread().getName() );
          profile.TreatThread.deadlockCheck3("b(Deallock3thread)", Thread.currentThread().getName() );
        }
    }
  }
```

(그림 7) 변환된 프로그램

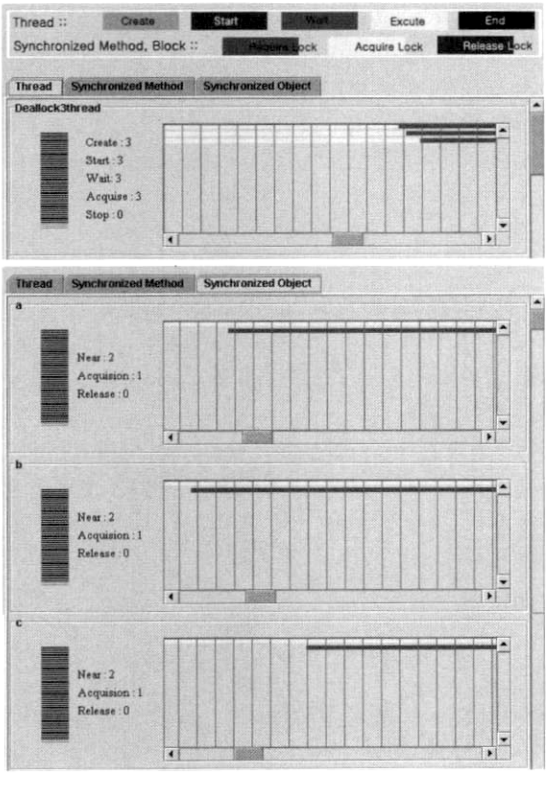
마다 각각을 모니터링하고 프로파일 하기 위한 코드를 (그림 7)과 같이 삽입한다. 모니터링과 프로파일을 위한 기본적인 기능은 별도의 라이브러리 profile.TreatThread 클래스 형태로 구현되었으며 실제로는 이 클래스가 제공하는 함수를 호출하는 코드가 삽입된다. 이 변환기는 또한 동기화에 의한 데드락을 탐지하기 위한 코드도 삽입되는데 각 동기화 객체에 대해서 동기화를 수행하여 점유하고 있는 쓰레드와 이를 기다리는 쓰레드들의 리스트(웨이팅 리스트)를 유지한다. 이러한 동기화 객체 점유와 웨이팅 리스트를 이용하여 루프가 생성되면 데드락이 발생한 것을 탐지할 수 있다.

변환된 EzSearch 프로그램의 일부분은 (그림 7)에서 볼 수 있다. 이 그림은 putMessage()가 동기화 함수임을 보여준다. 박스 안에 있는 코드는 변환기를 통해 삽입된 코드이다. 이것은 동기화 함수를 모니터링하고 실행과 관련된 정보를 저장한다. 변환된 프로그램은 Java 2 SDK에서 자동으로 실행된다. 변환된 코드는 실행 중에 실행 쓰레드와 동기화가 어떻게 처리되는지 모니터링 해 준다. 또한 이 시스템은 실행 쓰레드와 동기화 수와 같은 요약 정보를 프로그램 실행 직후에 프로파일 정보로 제공한다.

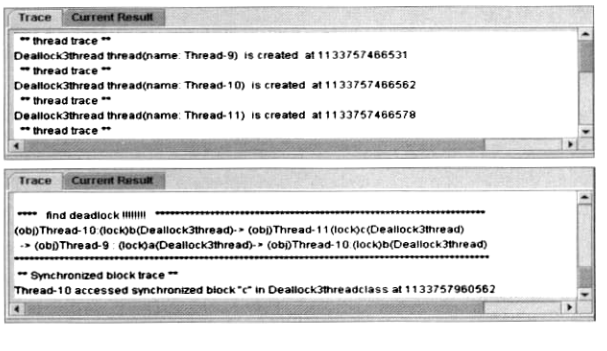
5. 실 험

본 연구에서는 Pentium 4 프로세서 기반 Window XP에서 SDK 1.4.2를 이용하여 시스템을 구현하였으며 이 시스템 상에서 벤치마크 프로그램을 대상으로 구현된 시스템의 효용성을 실험하였다. 먼저 예로 Deadlock3main 프로그램을 이용하여 실험하였다. 변환 후 데드락 프로그램을 실행했을 때 (그림 8)처럼 실행 쓰레드의 진행을 시각적으로 모니터링 할 수 있다. 실행중의 쓰레드의 진행을 시각적으로 보여주고 각 쓰레드는 실행 막대로 표현되었다. 막대의 빨간 색은 동기화를 위해 쓰레드가 기다리는 것을 나타낸다.

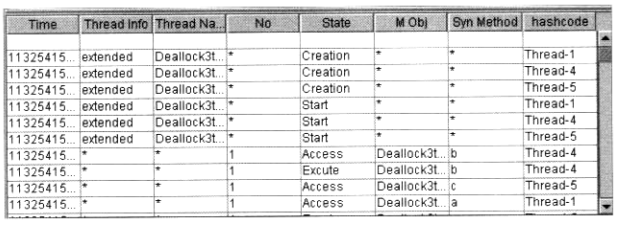
예를 들어 (그림 8)은 세 개의 모든 쓰레드가 기다리고 있다는 것을 보여준다. 하나의 쓰레드는 한 개의 객체를 동기화하였고 이를 다른 쓰레드가 기다리고 있다. (그림 9)는



(그림 8) 예제 프로그램의 트레이스



(그림 9) 데드락 발견



Time	Thread Info	Thread Na...	No	State	M Obj	Syn Method	hashcode
11325415...	extended	Deallock3t...	*	Creation	*	*	Thread-1
11325415...	extended	Deallock3t...	*	Creation	*	*	Thread-4
11325415...	extended	Deallock3t...	*	Creation	*	*	Thread-5
11325415...	extended	Deallock3t...	*	Start	*	*	Thread-1
11325415...	extended	Deallock3t...	*	Start	*	*	Thread-4
11325415...	extended	Deallock3t...	*	Start	*	*	Thread-5
11325415...	*	*	1	Access	Deallock3t...	b	Thread-4
11325415...	*	*	1	Excute	Deallock3t...	b	Thread-4
11325415...	*	*	1	Access	Deallock3t...	c	Thread-5
11325415...	*	*	1	Access	Deallock3t...	a	Thread-1

(그림 10) 데드락 프로그램의 프로파일

실행 동안에 데드락이 발생하였음을 보여준다. 프로그램이 종료했을 때 (그림 10)에서 보이는 것처럼 실행 쓰레드 수, 동기화 함수 호출, 동기화 블록의 접근의 수를 프로파일 정보로 제공한다.

본 연구에서는 5개의 벤치마크 프로그램을 가지고 실험하였다. 첫 번째 프로그램은 Read-Write 프로그램이다. 두 번째는 Deadlock3main 프로그램으로 실행 중에 데드락이 발생한다. 세 번째는 EzSearch 프로그램으로 입력 패턴과 매칭되는 파일을 찾아준다. 네 번째는 alarm 프로그램으로 모드에 따라 다른 시간을 보여주는 프로그램이다. 다섯 번째 프로그램은 ATApplet 프로그램으로 은행 계좌해서 출금하는 프로그램이다. <표 1>은 쓰레드 수, 실행 쓰레드 수, 동기화 함수의 수, 동기화 함수 호출 수, 동기화 블록의 수, 동기화 블록에 접근한 수를 보여준다.

〈표 1〉 실험결과 1

프로그램	쓰레드		동기화함수		동기화블록	
	종류	생성	종류	호출	종류	접근
Read-Write	2	6	5	13	3	8
Deadlock3main	1	3	0	0	3	3
EzSearch	2	20	2	25	0	0
Alarm	2	6	8	23	1	8
Atapplet	1	9	1	18	1	0

<표 2>는 코드 삽입 전과 모든 옵션을 선택한 상태에서 코드 삽입한 벤치마크 프로그램의 라인 수를 보여준다. 그리고 코드 삽입 전과 후의 프로그램 실행 시간을 보여준다. 몇 개의 옵션만을 선택하였다면 삽입 후 라인 수와 실행 시간을 줄어든다. Deadlock3main 프로그램에 있어서는 모니터링 시스템은 실행 중에 데드락을 발견한다.

〈표 2〉 실험결과 2

프로그램	줄수		시간(milliseconds)	
	변환전	변환후	변경전	변경후
Read-Write	221	293	1797	5719
Deadlock3main	70	160	데드락발생	
EzSearch	210	245	3282	4375
Alarm	336	462	12172	21235
Atapplet	272	342	21188	33703

6. 관련 연구

동적 프로그램 분석 기술은 실제 실행에 관한 정보를 제공하도록 연구되어 왔다[2, 6, 7, 9, 11]. J2ME Wireless Toolkit[7]과 AdaptJ[6]를 포함하여 자바를 위해 몇몇의 동적 분석 도구가 개발되었다. 최근 J2ME Wireless Toolkit은 JVMPI를 이용하여 메모리 사용, 예외 발생, 함수 호출 등을 추적할 수 있다. 또한 JVMPI는 프로그램에서 관심 있는 부분만을 효과적으로 추적하기에 어렵다. 왜냐하면 라이브러리를 포함한 모든 코드가 트레이스에 포함되기 때문이다.

AdaptJ는 동적 분석 도구로 실행시간동안 JVMPI를 이용하여 동적 정보를 얻는다. 그리고 파일로 저장한다. 실행 후에 동적 정보를 제공한다. 사용자는 몇몇의 옵션을 선택하여 유용한 정보를 선택할 수 있다. JFluid는 선 마이크로시스템에서 최근 개발되어진 분석 도구로 NetBeans IDE 에 통합되었다[9]. CPU, 메모리, 쓰레드 같이 실행 중인 자바 프로그램의 성능 데이터를 보여준다. 이것은 실행 직전에서 모니터링을 위하여 바이트 코드를 삽입한다. Concurrent Haskell Debugger는 Concurrent Haskell을 위한 디버깅 도구이다[11]. 이것은 데드락같은 병행성 문제를 시각적으로 보여줄 수 있다.

우리는 이전 연구로 실행시간에 예외 전파 및 처리 과정을 추적할 수 있도록 동적 예외 모니터링 시스템을 개발하였다[13]. 이 도구를 이용하여 프로그래머는 좀더 자세하게 예외 처리 과정을 살펴 볼 수 있고 이를 이용하여 좀더 효과적으로 예외를 처리할 수 있다.

본 연구의 쓰레드 모니터링 시스템은 코드 인라인 기법을 기반으로 하여 실행 쓰레드와 동기화의 진행과정을 시각적으로 효과적으로 보여줄 수 있다. 특히 정적분석 정보를 이용하여 제공될 수 있는 옵션을 사용자가 선택함으로써 관심있는 쓰레드와 동기화에만 초점을 맞춰 프로그램을 모니터링 할 수 있다. 또한 프로그램 실행 중에 발생하는 데드락을 탐지하여 보여줄 수 있다.

7. 결 론

본 연구에서는 멀티 쓰레드 자바 프로그램을 위한 동적 모니터링 시스템을 개발하였다. 이 시스템은 프로그래머가 멀티 쓰레드 프로그램의 진행 과정을 시각적으로 모니터링하도록 도와줄 수 있다. 또한 프로그래머는 옵션을 선택하여 관심 있는 쓰레드와 동기화만을 효과적으로 모니터링 할 수 있으며 프로그램 실행 중에 발생하는 데드락을 탐지할 수 있다.

본 연구는 두 가지 방향으로 확장할 것이다. 첫 번째는 변환된 프로그램의 실행 시간을 개선하는 것이다. 특히 시각화와 관련해서는 시스템의 효용성을 위해 실행 시간 개선이 필요하다. 두 번째는 프로그래머가 더 쉽게 자연스럽게 이해할 수 있도록 프로파일 정보와 쓰레드 모니터링의 시각화를 개선하는 것이다.

참 고 문 헌

[1] B. Bokowski, André Spiegel. Barat - A Front-End for Java Technical Report B 98-09 December 1998.

[2] B. Dufour, K. Driesen, L. Hendren and C. Verbrugge. Dynamic Metrics for Java, *Proceedings of ACM OOPSLA '03*, Octorber, 2003, Anaheim, CA.

[3] U. Erlingsson, *The Inlined Reference Monitor Approach to Security Policy Enforcement*, Ph.D thesis, Cornell University, January 2004.

[4] E. Gamma, R. Helm, R. Johnson and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995.

[5] J. Gosling, B. Joy, and G.Steele, *The Java Programming Language Specification,* Addison-Wesley, 1996.

[6] AdaptJ:A Dynamic Application Profiling Toolkit for Java, http://www.sable.mcgill. ca/bdufou1/AdaptJ

[7] Sun Microsystems, J2ME Wireless Toolkit for Java, http://java.sun.com

[8] Doug Lea, *Concurrent Programming in Java(TM) : Design Principles and Pattern* ,2nd Edition, Addusib Wesley.

[9] Sun Microsystems, Design of JFluid : A Profiling Technology and Tool Based on Dynamic Bytecode Instrumentation, 2003. 11.

[10] http://www.netbeans.org

[11] Jan Christiansen, Frank Huch: Searching for deadlocks while debugging concurrent haskell programs. ICFP 2004 : 28-39.

[12] Deepa Viswanathan, Sheng Liang: Java Virtual Machine Profiler Interface. IBM Systems Journal Vol.39, No.1, pp.82~, 2000.

[13] 오희정, 창병모, 신뢰성 높은 Java 프로그램 개발을 위한 예외 모니터링 시스템, 정보처리학회논문지 A, 제 12-A권 제6호, 2005년 12월.

문 세 원

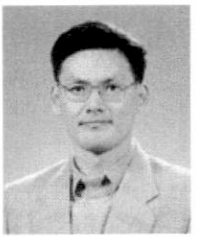
e-mail : wonsein@nate.com

2004년 숙명여자대학교 컴퓨터과학과 (이학사)

2006년 숙명여자대학교 컴퓨터과학과 (이학석사)

2006년~현재 삼성전자 연구원

관심분야 : 프로그래밍 언어 및 시스템



창 병 모

e-mail : chang@sookmyung.ac.kr

1988년 서울대학교 컴퓨터공학과(공학사)

1990년 한국과학기술원 전산학과(공학석사)

1994년 한국과학기술원 전산학과(공학박사)

1994년 한국전자통신연구원 박사후연구원

2002년 IBM Watson 연구소 방문과학자

2003년 Univ. of Pennsylvania 방문과학자

1995년~현재 숙명여자대학교 컴퓨터과학과 교수

관심분야 : 프로그래밍 언어 및 시스템, 유비쿼터스 소프트웨어