제18회 한국정보처리학회 추계학술발표대회 논문집 제9권 제2호 (2002.11)

병행 자바 프로그램 슬라이싱을 위한 다중쓰레드 종속성 그래프의 개선에 대한 연구

류희열*, 김은정**
*동의공업대학 컴퓨터정보계열
**부산외국어대학교 컴퓨터전자공학부
e-mail : hyryu@dit.ac.kr

A Study on Enhancement of Multithreaded Dependence Graphs for Concurrent Java Program Slicing

Hee-Yeol Ryu*, Eun-Jung Kim**
*Dept of Computer Information, Dong-Eui Institute of Technology
**Dept of Computer Engineering, Pusan University of Forein Studies

요 약

병행 자바 프로그램의 슬라이싱 방법은 Jianjun Zhao에 의해 제안된 다중 쓰레드 종속성 그래프를 이용하여 Susan Horwitz, Thomas Reps, David Binkly가 제안한 2단계 마킹 알고리즘을 적용하여 슬라이스를 계산한다. 다중 쓰레드 종속성 그래프를 이용하는 방법은 쓰레드 동기화 문장들 사이의 동기화 종속성과 서로 다른 쓰레드에 존재하는 공유객체들 사이의 통신 종속성 관계를 표현하여 병행 자바 프로그램의 슬라이스를 계산할 수 있는 것이다. 그러나 프로그램 종속성 그래프를 기반으로 하기 때문에 클래스 멤버 변수들에 대한 formal_in, formal_out, actual_in, actual_out정점들의 추가로 그래프의 복잡도가 증가하고 또한 부정확한 슬라이스 계산을 확인하였다. 따라서 본 논문에서는 병행 자바 프로그램 슬라이싱에서 정확한 슬라이스 계산을 위한 다중 쓰레드 종속성 그래프를 개선하여 제안한다. 제안하는 개선된 다중 쓰레드 종속성 그래프는 주어진 슬라이싱 기준에 대한 2단계 마킹 알고리즘을 적용한 결과 정확한 슬라이스 계산과 복잡도 개선을 확인하였다.

1. 서론

프로그램 슬라이싱은 자료흐름과 제어흐름을 분석하여 프로그램의 행위를 관심있는 특정한 부분으로 제한하는 기법으로 테스팅, 디버깅, 코드이해, 유지보수등에 유용하게 이용된다. 문장번호와 관심있는 변수의 쌍으로 구성된 슬라이싱 기준이 주어지면 자료흐름과 제어흐름 분석을 이용하여 자동적으로 슬라이스가 계산된다. 슬라이스는 원시 프로그램과 동일한 행위와 결과를 생성하고 실행 가능해야 한다는 특징이 있다[1]. 절차적 프로그램에 대한 슬라이싱은 소스코드를 프로그램 종속성 그래프(ProgramDependence Graph ; PDG)로 변환하여 Susan Horwitz, Thomas Reps, David Binkley가 제안하는 2단계 마킹 알고리즘을 적용하는 방법이 가장 정확한 슬라이스를 계산할 수 있다고 알려져 있다[2]. 따라서 많은 프로그램 슬라이싱에 대한 연구는 PDG를 기반으로 한 2단계 마킹 알고리즘을 적용한다. 최근 객체지향 프로그램의 슬라이싱에 대한 연구가 진행되면서 기존의 PDG 방법에 객체지향 개념을 표현할 수 있는 새로운 프로그램 표현방법들이 개발되고 있다. 이 또한 2단계 마킹 알고리즘을 적용하여 객체지향 프로그램에 대한 슬라이스를 계산하는 방법이다[3].

최근 객체지향 프로그래밍 언어 중에서 플랫폼에 독립적이고 다중쓰레드 기능을 지원하며 뛰어난 보

안성과 분산환경에서 동작하는 특징을 가진 자바프로그래밍 언어가 널리 사용되고 있다. 자바는 비동기적인 실행 쓰레드들 사이에 동기화를 제공하기 위해서 모니터(monotors)를 사용한다. 이러한 병행 자바 프로그램의 비결정적인 행위 때문에 병행 자바 프로그램의 예측, 이해, 디버깅은 순차적인 객체지향 프로그램보다 더욱 어렵다. 이러한 문제를 해결하기 위해서 Jianjun Zhao는 기존의 순차적인 객체지향 프로그램에 대한 표현방법으로 Larsen과 Harrold가 제안한 (System Dependence Graph ; SDG)에 쓰레드 종속성 그래프와 동기화 종속성 그래프와 통신 종속성 그래프를 추가하여 다중 쓰레드 종속성 그래프(Multithread Dependence Graph ; MDG)를 제안하였다. MDG를 이용한 병행 자바 프로그램의 슬라이싱은 2패스 마킹 알고리즘을 이용하여 슬라이스를 계산하였다[4].

MDG를 이용한 병행 자바 프로그램에 대한 슬라이싱은 SDG를 기반으로 하기 때문에 클래스의 각 메소드 호출정점에서는 actual_in과 actual_out정점을 사용하고 메소드 진입정점에서는 formal_in, formal_out정점을 사용하여 호출문맥을 표현한다.

MDG는 객체생성에 의한 멤버변수까지도 호출문맥으로 표현하기 때문에 불필요한 정점들의 추가가 발생하여 그래프의 복잡도를 높인다. 또한 객체생성에 의한 멤버변수의 정의와 사용에 대한 명시적이면서도 정확한 자료종속성 간선의 표현이 불가능하며 호출문맥으로도 정확하게 표현되지도 않는다.

따라서 본 논문에서는 정점의 개수를 최소화 하여 그래프의 복잡도를 낮추면서도 호출문맥을 정확하게 표현할 수 있고 또한 객체변수에 대한 정확한 자료 종속성 간선을 표현할 수 있도록 개선된 MDG(Enhanced Multithreaded Dependence Grah ; EMDG)를 제안한다. 제안된 EMDG를 이용한 병행 자바 프로그램에 대한 슬라이싱은 2단계 마킹 알고리즘을 적용한 결과 정확한 슬라이스가 계산됨을 확인하고 EMDG의 복잡도가 기존의 MDG보다 개선됨을 보인다.

2. EMDG

본 논문에서 제안하는 EMDG의 표현 방법은 다음과 같다.

2.1 객체변수의 표현

클래스에서 객체가 생성되면 각 객체는 멤버변수에 각각의 자료를 저장한다. 또한 같은 클래스에서 생성된 객체일지라도 이들 객체는 서로 독립적이다. 따라서 객체가 생성되면 각 객체의 멤버변수에 저장된 자료와 각 문장에서의 정의와 사용 관계를 정확하게 파악하여 자료 종속성 간선으로 연결한다.

객체생성을 위한 클래스의 생성자 호출이 발생하는 정점을 객체생성 정점이라 한다. 객체 생성 정점은 생성된 객체의 각 멤버변수와 멤버 종속선 간선으로 연결한다. 또한 클래스의 멤버변수 선언 정점과 생성된 객체의 멤버변수 정의 정점간의 자료 종속성 간선이 존재한다. 문장에서 정의되는 멤버변수는 해당문장에서 멤버변수 정점으로 자료 종속성 간선으로 연결하고 사용하는 문장정점에서는 멤버변수 정점에서 사용하는 문장정점으로 자료종속성 간선으로 연결한다.

2.2 호출문맥의 표현

기존의 MDG방법은 메소드 호출이 발생하면 호출정점에서는 실 매개변수 정점을 메소드 진입정점에서는 형식 매개변수 정점을 추가하며 객체의 멤버변수들도 동일하게 적용하는 방법이다. 그러나 제안하는 EMDG는 객체멤버변수는 매개변수 정점으로 추가하지 않는다. 객체 생성 정점에서 멤버변수 간선으로 연결되어 있는 객체멤버 변수 정점은 호출된 메소드 내부에서 정의관계가 존재하면 진입간선으로 연결하고 사용관계가 존재하면 진출간선으로 연결하여 자료 종속성을 나타낸다. 그리고 메소드 호출정점에서 실매개변수는 actual_in정점으로 연결하고 메소드 진입정점에서는 formal_in정점으로 연결하며 자료 종속성 간선으로 연결한다. 또한 formal_out정점에서 actual_out정점으로 자료 종속성 간선으로 연결한다. 따라서 호출문맥 표현에 객체변수에 대한 매개변수 정점을 제거하면서도 호출문맥을 정확하게 표현할 수 있다. 또한 정점의 개수를 줄여서 그래프의 복잡도를 낮출 수 있다.

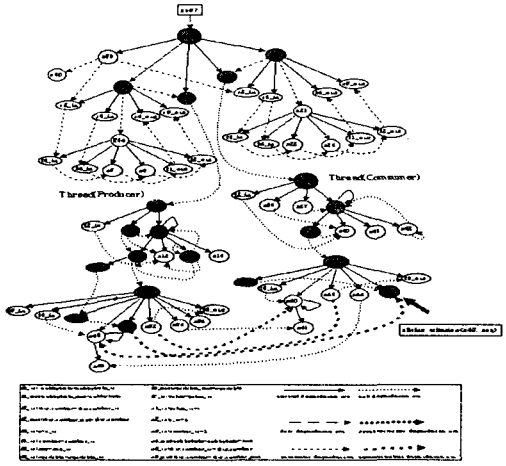
3. 슬라이싱 적용

MDG방법에서 사용한 병행 자바 프로그램을 이용하여 EMDG로 표현하고 2단계 마킹 알고리즘을 적용하여 정확한 슬라이스가 계산되는지를 비교 평가한다. (표 1)은 자바 예제 프로그램이다. (그림 1)은 슬라이싱 기준<S45, seq>에 대한 슬라이스된 MDG이다. 여기서 Consumer와 Producer 클래스의 생성자가 슬라이스로 계산되지 않는다. 그러나 본 논문에서 제안한 EMDG에 슬라이싱 기준<S45,

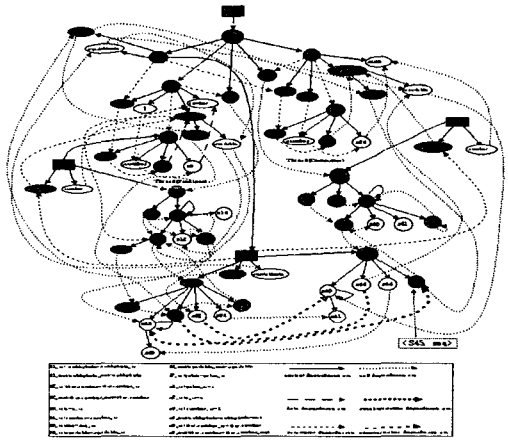
seq>을 적용하면 Consumer와 Producer 클래스의 생성자가 슬라이스로 계산된다. 슬라이스된 EMDG를 표현하면 (그림 2)와 같다. 슬라이스된 EMDG를 프로그램 코드로 변환하면 (표 2)와 같다.

(표 1) 병행 자바 프로그램

```
1  class Producer extends Thread {
2      private CubbyHole cubbyhole;
3      private int number;
4      public Producer (CubbyHole c, int number){
5          cubbyhole = c;
6          this.number = number;
7          }
8      public void run() {
9          int i = 0;
10         while (i < 10) {
11           cubbyhole.put(i);
12           System.out.println("Producer #" + this.number + "put:" + i);
13           sleep((int)(Math.random()*100));
14           i = i + 1;
15           }
16       }
17   }
18 class Consumer extends Thread {
19     private CubbyHole cubbyhole;
20     private int number;
21     public Consumer(CubbyHole c, int number) {
22         cubbyhole = c;
23         this.number = number;
24       }
25     public void run() {
26       int value = 0;
27       int i = 0;
28       while (i < 10) {
29         value = cubbyhole.get();
30         System.out.println("Consumer #" + this.number + "get:" + value);
31         sleep((int)(Math.random()*100));
32         i = i + 1;
33         }
34     }
35 }
36 class CubbyHole {
37     private int seq;
38     private boolean available = false;
39     public synchronized int get() {
40       while (available == false) {
41           wait();
42         }
43       available = false;
44       notify();
45       return seq;
46     }
47   public synchronized int put ( int value) {
48       while (available == true) {
49           wait();
50         }
51       seq = value;
52       available = true;
53       notify();
54       return seq;
55     }
56 }
57 class ProducerConsumerTest {
58   public static void main ( string[] args ) {
59       CubbyHole c = new CubbyHole();
60       Producer p1 = new Producer ( c, 1 );
61       Consumer c1 = new Consumer ( c, 1 );
62       p1.start();
63       c1.start();
64       }
65 }
```



(그림 1) 표1의 예제프로그램에 대한 MDG



(그림 2) 표1의 EMDG 표현

위 (그림 2)의 슬라이스된 EMDG를 코드로 변환하면 (표 2)와 같다.

(표 2) EMDG를 이용한 슬라이스

```
1  class Producer extends Thread {
2      private CubbyHole cubbyhole;
3
4      public Producer (CubbyHole c, int number){
5          cubbyhole = c;
6
7          }
8      public void run() {
9          int i = 0;
10         while (i < 10) {
11           cubbyhole.put(i);
12
13
14           i = i + 1;
15           }
16       }
17   }
```

```
18  class Consumer extends Thread {
19      private CubbyHole cubbyhole;
20
21      public Consumer(CubbyHole c, int number) {
22          cubbyhole = c;
23
24      }
25      public void run() {
26        int value = 0;
27        int i = 0;
28        while (i < 10) {
29          value = cubbyhole.get();
30
31
32          i = i + 1;
33        }
34      }
35  }
36  class CubbyHole {
37      private int seq;
38
39      public synchronized int get() {
40
41
42
43
44
45        return seq;
46      }
47    public synchronized int put ( int value) {
48
49
50
51        seq = value;
52
53
54        return seq;
55      }
56  }
57  class ProducerConsumerTest {
58    public static void main ( string[] args ) {
59        CubbyHole c = new CubbyHole();
60        Producer p1 = new Producer ( c, 1 );
61        Consumer c1 = new Consumer ( c, 1 );
62        p1.start();
63        c1.start();
64      }
65  }
```

4. 분 석

본 논문에서는 순차 객체지향 프로그램을 위한 SDG의 크기를 평가하기 위해서 Loren D. Larsen and Marry Jean Harrold가 제안한 방법을 사용하여 제안하는 EMDG 구성을 위한 메모리 공간 크기를 분석한다. 이 방법은 그래프의 크기를 정점들의 개수에 대한 상한 경계값으로 나타낸다. EMDG의 크기에 영향을 주는 요소는 (표 3)에 나타낸다.

(표 3) EMDG 크기에 영향을 주는 요소

항 목	정 의
V	단일 메소드에서 문장정점들의 최대 수
E	단일 메소드에서 간선들의 최대 수
P	메소드에 대한 형식인수의 최대 수
PV	= P + OV + CV
CV	클래스에 정의된 멤버변수들의 최대 수
OV	생성된 객체의 인스턴스 변수들의 최대 수
CS	단일 메소드에서 호출지역의 최대 수
TD	클래스 상속 트리의 깊이
M	클래스에서 메소드의 최대 수
C	클래스의 개수

Size(m) = O(V+CS*(1+TD*(2*PV))+2*PV)

EMDGSize(m) = O(V+CS*(1+TD*(2*P))+2*P)

기존의 MDG를 이용한 방법은 메소드 m에 대한 상한 경계값에 메소드의 개수를 곱해서 각 쓰레드 종속성 그래프(Thread Dependence Graph ; TDG)의 상한 경계값을 계산한다. 각각의 TDG의 상한 경계값들의 합을 계산하여 MDG의 상한 경계값으로 계산한다. 제안하는 방법의 메소드의 상한경계값과 클래스 종속성 그래프(Class Dependence Graph; CLDG)의 상한경계값과 전체 EMDG의 상한경계값은 아래와 같다.

Size(CLDG) = O(Size(m) * M)

$$\text{Size(EMDG)} = \sum_{i=1}^{C} Size(CLDG_i)$$

5. 결 론

본 논문에서는 병행자바 프로그램의 슬라이싱을 위한 확장된 EMDG를 제안하였다. 제안하는 방법은 객체가 생성되는 정점에서 인스턴스 변수 정점을 추가하고 이들 변수의 자료 종속성 관계를 정확하게 표현할 수 있다. 또한 인스턴스 변수에 대한 매개변수를 제거함으로써 매개변수 정점의 개수를 감소 시킨다. 따라서 그래프의 복잡도를 개선하였으며, 예제 프로그램의 EMDG에 대한 2단계 마킹 알고리즘을 적용하여 정확한 슬라이스가 계산됨을 확인하였다.

향후 연구과제로는 슬라이싱 툴의 구현과 동적 슬라이싱으로 확장을 등을 들 수 있다.

참고문헌

[1] Weiser, M. "Program Slicing." IEEE *Transactions on Software Engineering* 10,4(1884),pp.352-357.1984

[2] Suasn Horwits and Tomas Reps, and David Binkly,"Interprocedural Slicing Using Dependence Graph." ACM Transaction Programming Languages and Systems,Vol.12,No.1, January 1990,Pages 26-60.

[3] Loren D. Larsen and Marry Jean Harrold,"Slicing Object-Oriented Software." Proceedings of ICSE-18. pp.495-505, March 1996.

[4] Jianjun Zhao, "Multithreaded Dependence Graphs for Concurrent Java Programs," Proc.1999 International Symposium on Software Engineering for Parallel and Distributed System, pp.13-23, IEEE Society Press,May 1.