제30회 한국정보처리학회 추계학술발표대회 논문집 제15권 제2호 (2008. 11)

# 멀티스레드 프로그램의 디버깅을 위한 부분순서 수행 그래프 시각화

김혜림*, 김병철*, 전용기**
*경상대학교 컴퓨터과학과
**경상대학교 정보과학과
e-mail: cjswhddl@nate.com, {bckim, jun}@gnu.ac.kr

# Visualizing a Partial-Order Execution Graph for Debugging Multithreaded Programs

Hye-Rim Kim*, Byung-Chul Kim*, and Yong-Kee Jun**
*Dept. of Computer Science, Gyeongsang National University
**Dept. of Informatics, Gyeongsang National University

요 약

멀티스레드 프로그램의 효과적인 디버깅을 위해서는 스레드의 비결정성에 의해 야기되는 다양한 수행 양상의 직관적인 이해가 중요하다. 스레드 수행 양상을 시각화하는 기존의 기법들은 공유 변수의 접근 사건들 간의 부분 순서를 표현함으로써 시각적 복잡도가 높거나 이전 수행에서 결정된 락킹 순서를 표현하여 잠재되어 있는 다른 수행 양상에 대한 정보를 제공하지 못 한다. 본 논문은 프로그램 수행의 비결정적인 부분 순서는 락의 종류와 속성을 포함하는 코드 블록으로 시각화하고, 결정적인 부분 순서는 블록들을 연결하는 간선으로 시각화한다. 본 연구의 그래프는 플랫폼에 독립적인 Java Swing으로 구현하고 합성 프로그램을 사용하여 효과성을 실험한다.

## 1. 서론

멀티스레드 프로그램[4, 19, 9]의 효과적인 디버깅[16, 2, 14, 11, 8, 6]을 위해서는 스레드들 간의 비결정적인 수행으로 인한 다양한 수행 양상에 대한 이해가 중요하다. 멀티코어의 등장으로 멀티스레드 프로그램의 개발이 더욱 중요해지고 있지만 멀티스레드 프로그램을 이해하거나 디버깅하는 것은 여전히 어려운 과제이다. 스레드는 독립적으로 수행되면서 공유 자원에 대해서 경쟁하고 동기화되며 동적으로 생성 및 소멸된다. 또한 스레드의 비결정적인 수행은 동일한 프로그램을 동일한 입력으로 수행시키더라도 다른 수행 결과를 생산할 수도 있다. 이러한 비결정적인 수행은 이전 수행에서 야기된 부정확한 결과가 반드시 다음 수행에서 재생산된다는 것을 보장하지 못하므로 디버깅을 어렵게 한다.

멀티스레드 프로그램의 비결정적인 수행을 이해하기 위해서는 스레드들 간의 상호 작용을 이해하는 것이 중요하다. 스레드들 간의 상호 작용은 스레드들에 의해서 수행되는 사건들의 순서 관계[15, 7]로 정의할 수 있다. 멀티스레드 프로그램의 수행 사건들이 부분 순서를 나타내므로, 프로그램 수행 중에 발생한 사건들을 분석하여 사건들 간의 부분 순서 관계를 파악하는 것은 매우 복잡하고 수행 양상에 대한 다양한 변화를 이해하기가 어렵다. 따라서 멀티스레드 프로그램의 비결정적인 수행 양상의 효과적인 이해 방법으로 기존의 연구들은 스레드들에 의해서 수행된 동기화 사건들이나 공유 변수의 접근 사건들 간의 부분 순서를 시각화하여 사용자의 직관으로 프로그램의 비결정성을 탐지할 수 있도록 한다.

본 논문은 기존의 연구들이 가지는 시각적 복잡성과 잠재적인 다른 수행 양상에 대한 정보를 제공하지 못하는 문제점을 해결한다. 본 논문은 멀티스레드 프로그램의 수행 양상을 코드 블록들 간의 부분 순서를 시각화하는 기법을 제안한다. 본 논문은 고유한 루트 노드를 가지는 방향성있는 비순환 그래프를 이용한다. 그래프의 노드는 동일한 스레드에서 수행된 두 개의 연속적인 동기화 사건들 사이에 존재하는 명령어의 집합인 코드 블록으로 정의되고, 간선은 두 개의 서로 다른 노드들 사이에서 보장된 부분 순서를 의미한다. 프로그램 수행의 결정적인 부분 순서와 비결정적인 부분 순서에 대한 구분적인 표현을 위해서 본 논문은 비결정적인 부분 순서는 코드 블록 상에 표시를 하고, 결정적인 부분 순서는 블록들을 서로 연결하는 간선으로 시각화한다. 본 연구에서 제시하는 기법은 플랫폼에 독립적인 Java Swing으로 구현하고 합성 프로그램을 사용하여 효과성을 실험한다.

본 논문의 수행 그래프 시각화는 스레드들이 코드 블록을 수행할 때 가지는 락들의 정보를 시각적으로 명백하게 보임으로써 데드락의 탐지와 발생 원인에 대한 분석을 용이하게 한다. 하지만 프로그램의 수행 양상을 코드 블록으로 제한함으로써 공유 변수에 의존하여 발생하는 문제를 탐지하기는 어렵다.

본 논문의 구성은 다음과 같다. 2장에서는 멀티스레드 프로그램의 비결정적인 수행을 시각화하기 위한 기존 연구들에 대해서 살펴보고 3장에서는 멀티스레드 프로그램의 수행 구조에 대해서 기술한다. 4장에서는 본 논문에서 제안하는 수행 그래프의 생성과 시각화 기법을 소개한다. 5장에서는 수행 그래프의 구현과 실험 내용을 제시하고, 6장에서 결론 및 향후과제를 마지막으로 소개한다.

## 2. 관련 연구

멀티스레드 프로그램의 비결정적인 수행을 이해하기 위해서는 스레드들 간의 상호 작용을 이해하는 것이 중요하다. 멀티스레드 프로그램의 비결정적인 수행 양상의 효과적인 이해 방법으로 기존의 연구들은 스레드들에 의해서 사건

들 간의 부분 순서를 시각화하여 사용자의 직관으로 프로그램의 비결정성을 탐지할 수 있도록 한다.

스레드의 부분 순서 관계를 시각화하기 위해 Java[17, 12, 18, 1, 3], pthread[10, 20, 21, 5], 그리고 OpenMP[13] 프로그램 모델에서 최근까지 많은 연구가 행해져 왔다. 이들 기법은 스레드 수행 양상을 UML (Unified Modeling Language), Time-Process Diagram, 그리고 POEG (Partial Order Execution Graph) 등을 이용하여 표현한다.

UML (Unified Modeling Lanuage)의 Sequence Diagram을 이용해서 스레드의 순서 관계를 시각화하는 기법들[17, 18, 1, 3]은 스레드를 Active Object로 표현함으로써 스레드 객체와 구분하였고, y 축을 절대 시간 흐름하여 스레드의 진행 과정을 표현한다. Time-Process Diagram으로 스레드 수행 양상을 시각화하는 기법들[21, 5]은 y 축에 스레드들을 배치하고 x 축에 절대 시간 흐름에 따른 스레드의 상태를 표현한다. 이 기법들은 스레드들의 상태를 다양한 색상을 이용하여 표현하며, 스레드들 간의 락킹에 의한 동기화를 표현한다. POEG으로 스레드 수행 양상을 표현하는 기법들[10, 20, 12, 13]은 스레드들에서 수행되는 접근 사건과 동기화 사건들 간의 부분 순서를 표현하며, 논리적 시간 순으로 배열한다.

이들 기존의 연구들은 멀티스레드 프로그램의 비결정적인 수행 양상을 스레드들에 의해서 수행된 동기화 사건들이나 공유 변수의 접근 사건들 간의 부분 순서를 시각화함으로써 시각적 복잡성이 높거나 잠재적으로 가능한 다른 수행 양상에 대한 제한된 정보만을 제공한다.

## 3. 멀티스레드 프로그램의 수행 구조

스레드는 프로그램의 제어 흐름을 의미하며 멀티스레드 프로그램은 이러한 제어 흐름이 2개 이상 동시에 존재하는 프로그램을 말한다. 스레드들은 독립적으로 수행하면서 공유 자원에 대해서는 적절히 동기화되어서 협력한다. 스레드들은 병행하게 수행하므로 공유 자원에 대한 잘 못된 접근을 방지하기 위해서 임계구역을 사용한다. 임계구역은 한 번에 하나의 스레드만이 수행할 수 있도록 강제된 코드 영역이다. 또한 임계구역 내부에서 수행 중인 스레드가 다른 스레드에서 특정 사건이 발생할 때 까지 대기할 수도 있다.

스레드를 지원하는 다양한 프로그래밍 언어들의 문법적인 차이로 야기되는 혼란을 방지하기 위해서 본 논문은 스레드 사건들을 이용하여 프로그램의 수행을 설명한다. 이러한 스레드 사건들은 크게 2 종류로 구분된다. 하나는 스레드의 생성과 종료와 관련된 스레드 제어 사건들이고 다른 하나는 스레드들이 공유 자원을 안전하고 효율적으로 사용하기 위한 동기화 사건들이다.

스레드 제어 사건들로는 스레드의 시작을 나타내는 Thread Start (TS), 스레드의 종료를 나타내는 Thread Terminate (TT), 새로운 스레드를 생성하는 Thread Fork (TF), 마지막으로 다른 스레드가 종료할 때까지 대기하는 Thread Join (TJ)가 있다. 동기화 사건들로는 임계 구역에 진입하기 위해 락을 획득하는 Lock Acquire (LA), 임계구역을 탈출하면서 락을 해제하는 Lock Release (LR), 임계구역 내에서 다른 스레드에게 CPU 제어권을 양보하며 블럭되는 Wait (WT)와 임계구역 내에서 제어권을 양보하여 대기 중인 스레드를 해제하는 Notify (NT)가 있다.

[그림 1]은 본 논문에서 사용하는 스레드 사건들이 프로그램 수행 중에 수집된 내용을 보이고 있다. 이들 사건들은 스레드마다 별도의 파일에 저장된다. 첫 번째 열(timestamp)은 스레드에서 수행한 사건의 순서를 나타낸다. 사건의 순서는 논리적인 순서로 표현하며 매번 스레드 사건이 수행될 때마다 1씩 증가한다. 두 번째 열(type)은 스레드가 수행한 사건의 종류를 나타낸다. 세 번째 열(level)은 스레드가 현재 획득하고 있는 락의 개수를 의미한다. 네 번째부터 마지막 열까지는 스레드의 수행 사건에 종속적인 정보를 나타낸다. 네 번째 열(lid) 스레드 사건이 락과 관련된 경우에 락 식별자를 나타내며, 네 번째 열(stid)과 다섯 번째 열(stts)은 Thread Fork와 Thread Join에 의해서 해당 사건을 수행한 스레드 식별자와 스레드 사건 순서를 나타낸다.

## 4. 수행 그래프

본 논문에서 제안하는 수행 그래프는 프로그램의 코드 블록들 간의 부분 순서를 표현한다. 본 절에서는 이러한 수행 그래프의 생성하는 방법과 생성된 수행 그래프를 화면에 시각화하는 방법에 대해서 기술한다.

### 4.1 수행 그래프의 생성

본 논문은 스레드의 수행 양상을 코드 블록들 간의 부분 순서를 표현하는 수행 그래프로 시각화한다. 수행 그래프는 방향성을 가진 비순환 그래프이며 노드는 코드 블록으로 구성되며 간선은 코드 블록들 간의 부분 순서를 방향성으로 나타낸다. 코드 블록은 하나의 스레드에 의한 두 개의 연속된 스레드 사건들 사이에 존재하는 코드들이며 이들 간의 부분 순서는 Thread Fork, Thread Join 그리고 동일 스레드 내에서의 수행 순서에 의해서 결정적으로 생성되며, Lock Acquire, Lock Release, Wait, Notify에 의해서 비결정적으로 생성된다. 따라서 본 논문은 이러한 결정적인 부분 순서와 비결정적인 부분 순서에 대한 구분적인 표현을 위해서 비결정적인 부분 순서는 코드 블록 상에 표시를 하고, 결정적인 부분 순서는 블록들을 서로 연결하는 간선으로 표시한다.

코드 블록은 스레드 별로 주어진 추적 파일에서 두 개의 연속된 사건들을 분석하여 생성된다. 하나의 블록을 위해 사용된 두 개의 사건들 중에서 두 번째 사건은 다음 블록의 첫 번째 사건으로 사용된다. 블록은 식별자(id), 타입(type), 락 개수(level), 상위와 하위의 락의 종류(ulid, llid), 새로 생성된 블록 정보(ftid, fts), 합류하는 블록 정보(jtid, jts)를 포함한다.

블록의 식별자는 해당 코드 블록을 수행한 스레드 식별자와 스레드 내에서의 코드 블록의 순서로 이루어지며 전체 프로그램에서 고유한 값이다. 블록 타입은 두 개의 동기화 스레드 사건들의 조합으로 LockOpen은 첫 번째 사건이 Lock Acquire이며 두 번째 사건이 Lock과 관련이

| timestamp | type | level | lid | stid | stts |
|---|---|---|---|---|---|
| 1 | TS | 0 | 0 | 0 | 0 |
| 2 | TF | 0 | 0 | 1 | 1 |
| 3 | LA | 1 | L1 | 0 | 0 |
| 4 | NT | 1 | L1 | 0 | 0 |
| 5 | LR | 0 | L1 | 0 | 0 |
| 6 | TJ | 0 | 0 | 1 | 5 |
| 7 | TT | 0 | 0 | 0 | 0 |
| 1 | TS | 0 | 0 | 0 | 0 |
| 2 | LA | 1 | L1 | 0 | 0 |
| 3 | WT | 1 | L1 | 0 | 0 |
| 4 | LR | 0 | L1 | 0 | 0 |
| 5 | TT | 0 | 0 | 0 | 0 |

[그림 1] 프로그램 수행 중에 수집된 스레드 사건들

| id | type | level | ulid, llid | ftid, fts | jtid, jts |
|---|---|---|---|---|---|
| 0,1 | Normal | 0 | 0,0 | 1,1 | 0,0 |
| 0,2 | Normal | 0 | 0,0 | 0,0 | 0,0 |
| 0,3 | OpenNotify | 1 | L1, L1 | 0,0 | 0,0 |
| 0,4 | LockClose | 1 | 0, L1 | 0,0 | 0,0 |
| 0,5 | Normal | 0 | 0,0 | 0,0 | 0,0 |
| 0,6 | Normal | 0 | 0,0 | 0,0 | 1,5 |
| 1,1 | Normal | 0 | 0,0 | 0,0 | 0,0 |
| 1,2 | LockOpen | 1 | L1,0 | 0,0 | 0,0 |
| 1,3 | WaitClose | 1 | L1, L1 | 0,0 | 0,0 |
| 1,4 | Normal | 0 | 0 | 0,0 | 0,0 |

[그림 2] 코드 블록 정보

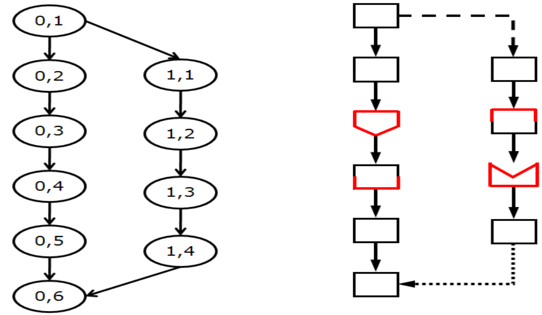
없는 사건을 나타낸다. LockClose는 첫 번째 사건은 락과 관련이 없으나 두 번째 사건은 Lock Release인 블록을 의미한다. OpenNotify는 첫 번째 사건이 Lock Acquire이고 두 번째 사건이 Notify 인 블록이고, WaitClose는 첫 번째 사건이 Wait이고 두 번째 사건이 Lock Release인 블록을 나타낸다. 마지막으로 LockWrapped는 첫 번째 사건이 Lock Open이고 두 번째 사건이 Lock Release인 블록이다. 락 개수는 해당 블록이 획득하고 있는 락의 총수를 의미하며, 상위와 하위의 락 종류는 블록의 첫 번째 사건의 락과 두 번째 사건의 락 식별자를 나타내며 락 식별자가 없는 경우는 0으로 한다. 새로 생성된 블록 정보는 해당 블록의 하위 사건인 Thread Fork에 의해서 생성되는 스레드의 식별자 및 타임스탬프를 의미한다. 합류하는 블록 정보는 해당 블록의 첫 번째 스레드 사건인 Thread Join에 의해서 합류되는 스레드의 식별자 및 타임스탬프를 나타낸다. [그림 2]는 [그림 1]의 스레드 사건으로 생성된 코드 블록들의 정보를 보이고 있다.

수행 그래프는 이들 코드 블록들을 노드로 한다. 간선은 동일한 스레드 내의 순차적 부분 순서와 다른 스레드와의 Thread Fork에 의한 부분 순서, 그리고 Thread Join에 의한 부분 순서를 표현한다. 하나의 블록에서 이들 부분 순서들의 정보는 각각의 리스트로 관리된다. 순차적으로 연결되는 코드 블록은 현재 블록 식별자 <tid, ts>에서 <tid, ts+1>의 값을 식별자로 하는 블록이 된다. Thread Fork에 의한 부분 순서는 현재 블록의 <ftid, fts> 정보와 일치하는 블록 식별자를 가지는 블록이 된다. Thread Join에 의한 부분 순서 관계의 블록은 현재 블록의 <jtid, jts>에서 <jtid, jts-1>의 값을 식별자로 하는 블록이 된다. 모든 블록에 대해서 이러한 부분 순서 관계의 블록을 찾아서 블록이 존재한다면 그 블록의 노드와 연결을 하고 존재하지 않는다면 NULL 값을 배정한다.

### 4.2 수행 그래프의 시각화

본 논문은 스레드 프로그램의 비결정적인 수행 양상에 대한 직관적인 탐지를 위해서 수행 그래프를 시각화한다. 생성된 수행 그래프를 시각화하기 위해서 노드는 기본적으로 검은색의 정사각형으로 표현하고 간선은 부분 순서의 종류에 따라서 화살표가 있는 검은색 선으로 표현한다.

노드에서 유지하는 락 정보의 표현은 사각형의 아래와 위에는 락 식별자에 해당하는 고유한 색상을 사용한다. Wait와 Notify 스레드 사건을 포함하고 있는 스레드는 "V" 자 모양으로 해당 락 식별자 색상을 이용하여 표현한다. 블록들 간의 결정적인 부분 순서를 나타내는 간선은 동일한 스레드 내에서의 부분순서는 실선으로, Thread Fork에 의한 부분 순서는 쇄선으로, Thread Join에 의한 부분 순서는 점선으로 나타내며 부분 순서의 의미를 화살표 방향으로 표현한다.




(a) 내부적 표현 (b) 시각적 표현

[그림 3] 수행 그래프

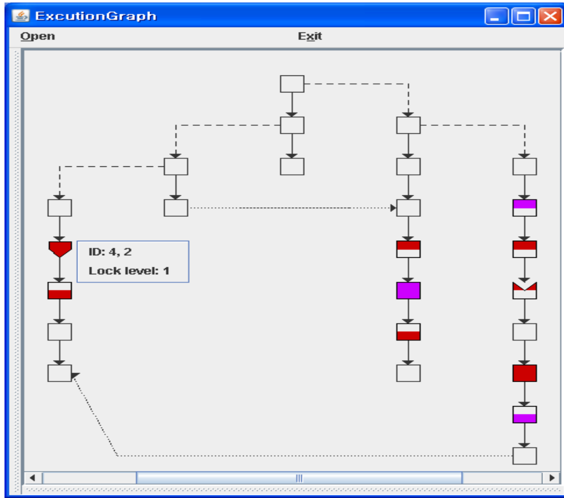
수행 그래프의 디스플레이는 화면에 배치될 스레드 위치와 블록 위치의 결정이 필요하다. 스레드의 위치는 두 개 이상의 스레드가 하나의 열을 차지하지 않도록 한다. 메인 스레드가 화면의 정중앙에 표현이 되며, 메인 스레드에서 홀수 번째로 생성되는 스레드는 메인 스레드의 오른쪽에 배치되고, 짝수 번째로 생성되는 스레드는 메인 스레드의 왼쪽에 배치된다. 이때 가장 최근에 생성된 스레드가 메인 스레드와 가장 가까운 위치에 배치되도록 한다. 메인 스레드 이외의 스레드에서 생성된 스레드들은 메인 스레드에서 부모 스레드의 전개 방향과 동일하게 왼쪽과 오른쪽으로 배치된다. 여기서도 하나의 스레드에서 가장 최근에 생성된 스레드가 부모 스레드와 가장 근접하게 배치한다. 스레드의 블록들은 동일한 열에 발생된 순서대로 위에서 아래로 배치되며, Thread Fork로 연결된 두 블록들은 먼저 발생한 블록이 뒤에 발생한 블록보다 위에 배치된다. 이러한 수행 그래프의 배치는 화면 중앙을 기준으로 서로 균등하게 스레드를 배치할 수 있으므로 구조적인 안정감과 시각적인 균형감을 제공한다. [그림 3]은 [그림 2]의 블록 정보를 이용하여 생성한 수행 그래프의 내부적인 표현과 수행 그래프를 시각화한 것을 보이고 있다.

## 5. 구현 및 실험

본 논문에서 제안하는 수행 그래프는 Java Foundation Class (JFC)의 일부인 Swing을 이용해서 구현한다. Java의 플랫폼 독립적인 수행과 Swing의 유연한 그래픽 인터페이스는 구현된 수행 그래프를 시스템과 컴퓨팅 환경의 제한을 벗어나게 한다.

수행 그래프를 구현하기 위해서 본 논문은 추적 파일 분석기, 수행 그래프 생성기, 수행 그래프 시각기 모듈을 구현하였다. 추적 파일 분석기는 프로그램 수행 중에 수집된 동기화 명령들을 분석하여 코드 블록을 생성하는 모듈이고, 수행 그래프 생성기는 생성된 코드 블록들을 노드와 간선으로 분류하여 수행 그래프를 생성하는 모듈이다. 마지막으로 수행 그래프 시각기는 생성된 수행 그래프를 심벌로 대체하여 화면에 배치한다.

수행 그래프의 실험을 위해서 본 논문은 합성 프로그램(synthetic program)을 개발하여 해당 소스 파일에 동기화 수집 명령을 삽입하여 추적 파일을 생성하였다. [그림 4]의 수행 그래프는 생성된 추적 파일을 시각화하여 사용자가 선택한 노드의 정보를 보이고 있다. 수행 그래프는 해당 프로그램이 전체 5개의 스레드로 구성되고 스레드들은 2개의 락을 사용하여 서로 동기화되고 있음을 보이고 있다. 또한 두 개의 스레드가 다른 두 개의 스레드가




[그림 4] 구현된 수행 그래프의 시각화

각각 종료한 후에 종료함을 알 수 있다. 특히 이 수행 그래프는 화면 오른쪽의 두 개의 스레드가 두 개의 락을 서로 다른 순서로 사용하고 있음으로 보여준다. 이는 해당 프로그램에서 이전 수행에서는 발생하지 않았지만 다음 수행에서는 데드락(deadlock)이 발생할 수 있음을 암시한다. 이러한 정보의 제공은 프로그래머로 하여금 오류를 발생하지 않은 멀티스레드 프로그램의 수행에서 조차도 다른 수행에서 발생할 수 있는 오류에 대한 디버깅을 가능하게 한다.

## 6. 결론 및 향후 과제

본 연구는 멀티스레드 프로그램에서 스레드들 간의 부분순서를 표현하는 수행 그래프를 시각화하여 사용자가 프로그램의 수행을 직관적으로 탐지하여 발생가능한 오류에 대해서 디버깅을 가능하게 하는 기법을 제안했다. 본 논문의 수행 그래프는 프로그램 수행 중에 발생하는 사건들 보다는 오히려 프로그램 수행 블록을 표현함으로써 그래프의 시각적 복잡성을 감소시켰다. 또한 프로그램 수행의 비결정적인 부분순서는 락의 종류와 속성을 포함하는 코드 블록으로 시각화하고, 결정적인 부분순서는 코드 블록들을 연결하는 간선으로 표현함으로써 이전 수행에서 발생하지 않았지만 다음 수행에서는 발생할 수 있는 프로그램의 수행 양상에 대한 이해를 가능하게 한다.

본 연구의 향후과제로는 다양한 스레드 환경에서 개발된 멀티스레드 프로그램의 수행 양상을 수집하는 기법과 소스 코드를 연계하는 기법을 개발하여 멀티스레드 프로그램의 시각적 디버깅 도구를 개발하는 것이다.

## 참고문헌


[1] Artho, Cyrille, Klaus Havelund, and Shinichi Honiden, "Visualization of Concurrent Program Executions," NII technical report #NII-2007-006E, Extended version of SACT paper, May 2007.

[2] Marcus, Aaron, Chris DiGiano, and Ron Baecker, "Software Visualization for Debugging," Communications of the ACM, 40(4):44-54 , ACM, April 1997.

[3] Bi, Yaodong, and John Beidler, "A Visual Tool for Teaching Multithreading in Java," Journal of Computing Sciences in Colleges, 22(6):156-163, Consortium for Computing Sciences in Colleges, June 2007.

[4] Birrell, D. Andrew, "An Introduction to Programming with Threads," Technical Report SR-35, Digital Equipment Corporation, Jan. 1989.

[5] Carr, Steve, Jean Mayo, and Ching-Kuang Shene, "ThreadMentor: A Pedagogical Tool for Multithreaded Programming," Journal on Education Resources in Computing (JERIC), 3(1):1-30, ACM, March 2003.

[6] Diehl, Stephan, Software Visualization: Visualizing the Structure, Behavior, and Evolve of Software, Springer, 2007.

[7] Fidge, C. J., "Partial Orders for Parallel Debugging," SIGPLAN/SIGOPS Workshop on Parallel and Distributed Debugging, pp. 183-194, ACM, May 1988.

[8] Gatlin, Su Kang, "Trials and Tribulations of Debugging Concurrency," ACM Queue, 2(7):66-73, ACM, Oct. 2004.

[9] Gosling, James, Bill Joy, Guy Steele, and Gilad Bracha, The Java Language Specification, Third Edition, Prentice Hall PTR, 2005.

[10] Helmbold, P. David, Charles E. McDowell, and Jian-Zhang Wang, "TraceViewer: A Graphical Browser for Trace Analysis," Technical Report USCS-CRL-90-59, University of California at Santa Cruz, Santa Cruz, CA, 1990.

[11] Huselius, Joel, "Debugging Parallel Systems: A State of the Art Report," Technical Report MRTC 63, Department of Computer Science and Engineering at Mälardalen University, Sept. 2002.

[12] Jackson, David, "Visual Debugging of Multithreaded Java Programs," Proc. of the IEEE 2001 Symposia on Human Centric Computing Languages and Environments (HCC'01), 340 - 341, IEEE, September 2001.

[13] Kim, Jeong-Si, Dong-Gook Kim, and Yong-Kee Jun, "Scalable Visualization for Debugging Races in OpenMP Programs," Proc. of the 3rd Int'l Conf. on Communications in Computing (CIC), 259-265, Las Vegas, Nevada, June 2002.

[14] Kraemer, Eileen, "Visualizing Concurrent Programs," Software Visualization: Programming as a Multimedia Experience, 237-258, The MIT Press, Cambridge, Massachusetts, 1998.

[15] Lamport, L., "Time, Clocks, and the Ordering of Events in a Distributed System," Communications of the ACM, 21(7): 558-565, ACM, July 1978.

[16] McDowell, Charles E., and David P. Helmbold, "Debugging Concurrent Programs," ACM Computing Surveys (CSUR), 21(4):593-622, ACM, Dec. 1989.

[17] Mehner, Katharina, and Annika Wagner, "Visualizing the Synchronization of Java-Threads with UML," Proc. of the 2000 IEEE Int'l Symposium on Visual Languages (VL'00), 199-206, IEEE, September 2000.

[18] Roychoudhury, Abhik, "Depiction and Playout of Multi-threaded Program Executions," Proc. of the 18th IEEE Int'l Conf. on Automated Software Engineering (ASE'03), 331 - 336, IEEE, Oct. 2003.

[19] Sanden, Bo, "Coping with Java Threads," IEEE Computer, 37(4):20-27, IEEE, April 2004.

[20] Zernick, Dror, Marc Snir, and Dalia Malki, "Using Visualization Tools to Understand Concurrency," IEEE Software, 9(3):87-92, IEEE, May 1992.

[21] Zhao, Qiang A., and John T. Stasko, "Visualizing the Execution of Threads-based Parallel Programs," Technical Report GIT-GVU-95-01, College of Computing, George Institute of Technology, January 1995.