# 내부적 비결정성을 가진 공유 메모리 프로그램의 잠재적 경합 탐지†

정민섭,* 김영주,** 하옥균,*** 전용기***
*(주)피더스테크, **한국정보통신대학교 IT공학부, ***경상대학교 정보과학과
e-mail: msjung@fidestech.co.kr, yjkim@icu.ac.kr, jassmin@race.gnu.a.kr, jun@gnu.ac.kr

# Potential Races Detection in Shared-Memory Programs with Internal Nondeterminism

Min-Sub Jung,* Young-Joo Kim,** Ok-Kyoon Ha,*** Yong-Kee Jun***
*FIDESTECH. Co., Ltd.
**School of Engineering, Information and Communications University
***Dept of Information Science, Gyeongsang National University

요 약

임계구역을 가진 공유 메모리 기반의 병렬 프로그램에서 발생하는 경합은 프로그래머가 의도하지 않은 비결정적인 수행 결과를 초래하므로 반드시 디버깅해야 한다. 이러한 경합을 수행 중에 탐지하는 기존의 기법들은 임계구역의 실행순서에 의해서 발생하는 내부적 비결정성이 존재하지 않는 프로그램에 대해서만 경합의 존재를 검증할 수 있다. 본 논문에서는 내부적 비결정성을 가진 프로그램에 존재하는 비결정적 접근사건을 정적으로 분석하고, 이 정보를 이용하여 수행 중에 경합을 탐지함으로써 잠재되어 있는 경합까지 탐지할 수 있는 도구를 제안한다. 제안한 도구는 비결정성이 포함된 합성프로그램과 공인된 OpenMP 벤치마크 프로그램인 Microbenchmark를 이용하여 경합 검증이 가능함을 보인다.

## 1. 서론

병렬 프로그램은 병행하게 수행하는 스레드간의 동기화가 필수적이다. 하지만 잘못된 동기화의 사용으로 인해 경합이라는 치명적인 오류가 발생할 수 있다. 경합[7]은 병행하게 수행하는 스레드가 적절한 동기화 없이 적어도 하나 이상의 쓰기 사건으로 공유메모리에 접근할 때 발생하며 이러한 경합은 프로그래머가 의도하지 않은 비결정적인 수행 결과를 초래하므로 반드시 탐지 되어야 하며 디버깅의 대상이 된다.

수행 중 경합 탐지 기법[4,8,9,10,11]을 이용하여 임계구역을 가진 프로그램에서 적어도 하나의 경합을 탐지하기 위해서는 SISE(Single Input, Single Execution) 속성[1,4]을 만족하여야 한다. 이 속성을 만족하지 못한다면 경합검증을 위해서 $N!$ 만큼의 프로그램 수행양상이 필요하다. 여기서 SISE 속성은 주어진 입력에 대하여 한번의 수행 양상에 대해서만 경합을 탐지하는 것을 의미하고, $N$은 최대병렬성을 의미한다. 이러한 기존의 경합탐지 도구들은 수행 중에 발생한 접근 사건정보만을 이용하여 경합을 탐지하므로 임계구역에 의한 내부적 비결정성이 존재하는 프로그램에 대해서는 SISE 속성을 만족하지 못한다. 따라서 본 논문에서는 내부적 비결정성을 가진 프로그램에 존재하는 비결정적 접근사건을 정적으로 분석하고, 이 정보를 이용하여 수행 중 경합을 탐지함으로써 내부적 비결정성이 존재하는 프로그램에서도 주어진 입력에 대하여 한 번의 수행만으로도 잠재되어 있는 경합까지 탐지할 수 있는 도구를 제안한다.

2절에서는 경합과 기존 도구의 문제점을 소개하고 3절에서는 잠재적 경합을 탐지하기 위해서 제안된 기법들을 설명한다. 그리고 4절에서는 제안된 기법을 이용하여 구현한 도구로 경합 검증이 가능함을 보인다. 마지막으로 5절에서는 결론 및 향후 과제에 기술한다.

†본 연구는 한국 학술진흥재단의 국제공동연구-협력기관 지정 사업으로 수행되었음.

## 2. 연구배경

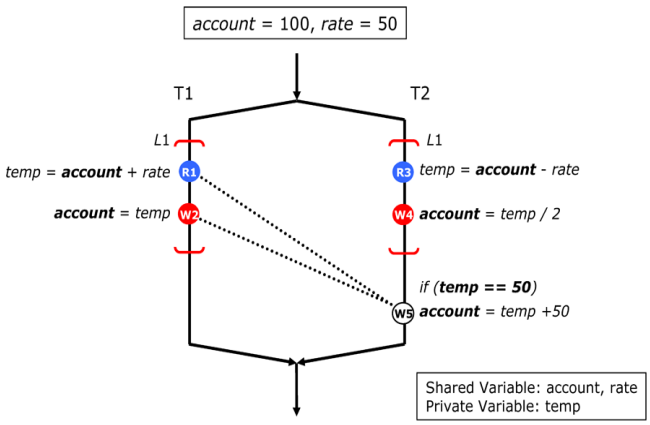
본 절에서는 POEG(Partial Order Execution Graph)[3]을 이용하여 병렬 프로그램에서 발생하는 경합에 대해서 설명하고, 임계구역을 가진 병렬 프로그램에서 경합을 탐지하는 기존 도구에 대한 문제점을 기술한다.

### 2.1 경합탐지

[그림 1]은 병렬 프로그램의 소스코드를 기반으로 작성된 POEG이다. POEG은 병렬 프로그램의 논리적 순서관계를 그래프로 나타낸 것이다. POEG에서의 정점(vertex)은 병렬 스레드의 생성이나 종료 명령에 의해 병렬 스레드의 포크(fork) 및 조인(join)되는 지점을 의미하며, 임의의 정점에서 시작하는 간선(arc)은 그 정점으로부터 수행되는 스레드의 블록을 표시한다. 이러한 POEG을 통해 쉽게 스레드의 병행성 관계를 파악할 수 있다. POEG에서 R 심볼과 W 심볼은 각각 그 스레드에서 발생한 공유변수 account에 대한 접근사건을 의미하며 ⊃ 심볼은 임계 구역의 시작과 끝을 의미한다.

[그림 1]에서 T1 스레드의 임계구역이 수행되고 T2의 임계구역이 수행되면 *account*의 값이 50이고, T2 스레드의 임계구역이 수행되고 T1 스레드의 임계구역이 수행되면 *account*의 값이 75가 된다. 하지만 프로그래머가 의도하지 않은 비결정적인 수행결과가 나올 수 있다. 예를 들어, T2 스레드의 임계구역이 먼저 록 변수 *L1*을 획득하여 *account*에 대한 읽기접근사건이 발생한 다음에 쓰기접근사건이 발생하기 전에 T1 스레드의 읽기접근사건이 발생하게 되면 *account*의 값이 150이 된다. 이러한 발생의 원인은 T1 스레드의 읽기접근사건과 T2 스레드의 쓰기접근사건 사이에 경합(race)이 존재하기 때문이다. 경합은 병렬로 수행하는 두 스레드가 적어도 하나 이상의 쓰기 사건으로 적절한 동기화 없이 같은 메모리 영역을 접근할 때 발생한다.

이러한 경합을 수행중에 탐지하기 위해서는 병렬로 수행하




[그림 1] 내부적 비결정성에 의한 잠재적 경합

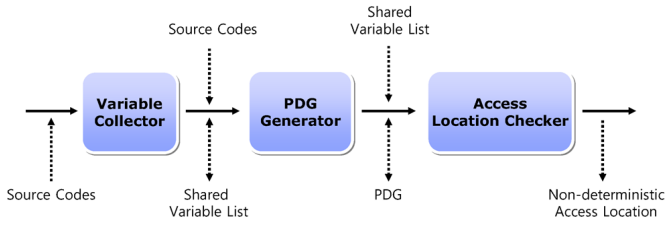
는 스레드간의 논리적 순서관계를 생성하는 레이블링 기법[6]과 각 스레드에서 발생한 공유변수에 대한 접근사건 정보를 저장하는 "Access History"의 유지정책으로 경합을 보고하는 탐지프로토콜[4,8,10] 기법이 필요하다.

## 2.2 기존의 경합탐지 기법

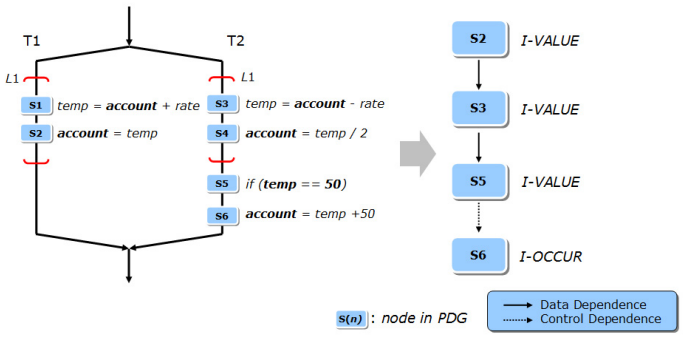
[그림 1]에서 각 스레드에 존재하는 임계구역의 수행 양상에 따라 수행되는 명령어 문장들이 달라질 수 있다. T2 스레드의 임계구역이 수행한 뒤 T1 스레드의 임계구역이 수행하게 되면 private 변수인 *temp*의 값이 50이 된다. 따라서 if문의 조건을 만족하게 되어 공유변수 *account*에 대한 쓰기사건 W5가 발생하게 된다. 하지만 T2 스레드의 임계구역이 먼저 수행한 뒤 T1 스레드의 임계 구역이 수행하게 되면 *temp*의 값이 100이 되기 때문에 if문에 내포 되어 있는 쓰기사건 W5는 발생하지 않는다. 즉 임계구역의 수행순서에 의한 내부적 비결정성이 존재한다. 또한, 공유변수 *account*에 대한 쓰기사건 W5도 비결정적 접근사건이다. 비결정적 접근사건을 가진 프로그램에서 기존의 기법[1,4]을 이용하여 경합을 탐지할 경우 T1 스레드의 임계구역이 먼저 수행 되었다면 W5에 대한 접근 정보가 없기 때문에 [그림 1]에 보인 것처럼 {R1-W5}와 {W2-W5} 사이의 경합은 보고하지 못한다. 왜냐하면 내부적 비결정성이 존재하지 않는 프로그램에 대해서만 SISE 속성을 만족하기 때문이다. 따라서 내부적 비결정성을 가진 프로그램에서 기존의 기법은 경합을 탐지하지 못한다. 그러나 비결정적 접근사건에 의한 잠재적 경합이 존재할 수 있으므로 이러한 잠재적 경합을 탐지하기 위해서는 $N!$ 만큼의 프로그램 수행이 필요하다. 여기서 $N$은 최대병렬성을 의미한다. 예를 들어, [그림 1]에서 T1 스레드의 임계구역이 수행한 후에 T2 스레드의 임계구역이 수행했을 경우와 T2 스레드의 임계구역이 수행한 후에 T1 스레드의 임계구역이 수행했을 경우의 수행양상을 모두 고려하여야만 경합의 검증이 가능하다.

## 3. 잠재적 경합 탐지

본 절에서는 잠재적 경합을 탐지할 수 있는 도구에 대한 모듈




[그림 2] 내부적 비결정성 분석을 위한 모듈



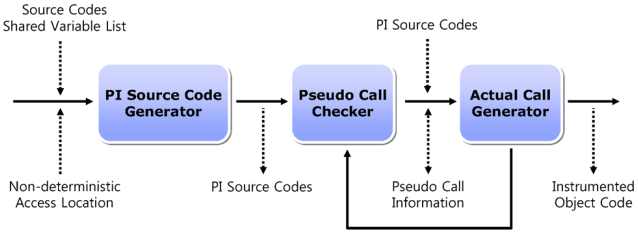

[그림 3] PDG를 이용한 비결정적 접근사건

에 대해서 설명한다. 첫 번째, 비결정적 접근사건의 위치 정보를 파악하는 모듈과 두 번째, 분석된 정보를 이용하여 수행중에 경합을 탐지할 수 있도록 감시코드를 생성하는 모듈에 대해서 설명한다.

## 3.1 내부적 비결정성 분석

비결정적 접근사건의 위치정보를 분석하기 위해서 [그림 2]와 같은 모듈들을 가진다. "Variable Collector"는 대상 프로그램에서 사용된 전역변수나 지역변수들에 대한 정보를 수집하는 모듈이고, "PDG Generator"는 프로그램의 명령어 문장이나 조건식을 노드로 표시하며, 노드간의 데이터 의존성과 제어 의존성을 생성하는 모듈이다. 그리고 "Access Location Checker"는 수집된 변수의 정보와 생성된 PDG(Program Dependency Graph)[5]를 이용하여 Dinning이 제안한 Forward-Slice 알고리즘[5]을 이용하여 비결정적 접근사건의 위치를 검사하는 모듈이다.

[그림 3]은 PDG를 이용하여 Forward Slicing 알고리즘으로 비결정적인 접근사건의 위치를 파악하는 예를 보인 것이다. T1 스레드의 S1 노드에서 변수 *temp*의 값은 S2 노드의 변수 *account*의 값에 영향을 미친다. 이때 이 두 노드 사이에는 데이터 의존성이 존재한다. 그리고 T2 스레드의 S5 노드에서 조건문에 따라 S6노드의 수행 여부가 결정된다. 이때 이 두 노드 사이에는 제어 의존성이 존재한다. 임계구역 내에서 발생한 공유변수에 대한 쓰기접근사건은 내부적 비결정성의 발생 원인이 된다. S2 노드와 S4 노드의 공유변수 *account*는 T1 스레드와 T2 스레드에 존재하는 임계구역의 수행 순서에 따라 비결정적인 값을 가진다. 그리고 S6 노드의 경우 S5 노드에 의해 비결정적으로 수행되며 S6 노드의 공유변수 *account*에 대한 접근사건은 비결정적으로 발생한다. 따라서 생성된 PDG를 이용하여 Dinning이 제안한 Forward Slice 알고리즘을 수행하면 S1 노드부터 S5 노드까지 I-VALUE로 설정이 되고, S6 노드는 I-OCCUR로 설정된다. 여기서 S6 노드에 직접적으로 영향을 주는 I-VALUE 노드들을 [그림 3]과 같이 S2, S3, 그리고 S5이다. I-OCCUR로 설정된 S6 노드의 *account*에 대한 접근사건 정보를 PDG에 추가하여 유지한다. I-VALUE는 임계구역 내에




[그림 4] 감시코드 생성을 위한 모듈

```
1. #progma omp parallel for shared (account,rate) private(i,temp)
2. for(i=0; i<2; I++) {
3. label_main1 = (Label_NR *)NR_Fork(label_main0,i,1,0,2,1,7)
4. if(i==0) {
5.    #pragma omp critical(L1)
6.    {
7.       NR_Add_Lock(label_main1, "L1", 10);
8.       CSR_Checker(label_main1, SV0, 12);
9.       CSR_Checker(label_main1, SV1, 12);
10.       temp = account + rate;
11.       ...
12. } else if(i==1) {
13.      ...
14.     if(temp == 50) {
15.        account = temp + 50;
16.     }
17.     D_W_Checker(label_main1, SV0, 24);
18. }}
```

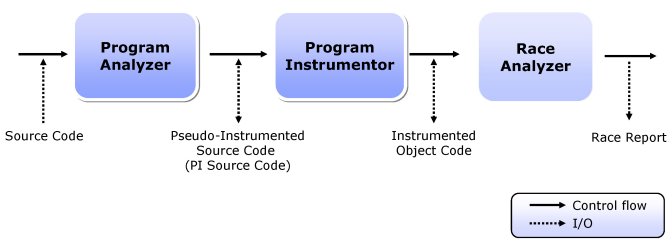
[그림 5] OpenMP 프로그램에 대한 변형된 원시 프로그램

서 공유변수나 지역변수에 대한 쓰기접근사건이 발생한 것을 의미하고, I-OCCUR는 명령어 문장이 비결정적으로 수행되거나 변수의 참조가 달라지는 것을 의미한다.

### 3.2 감시코드 생성

내부적 비결정성을 수행중에 탐지하기 위해서는 원시프로그램이나 바이너리 코드에 감시코드를 추가해야 된다. 본 연구에서는 원시프로그램에 감시코드를 추가하는 방법을 사용한다. [그림 4]는 [그림 2]에서 수집된 변수 리스트와 비결정적 접근사건에 대한 정보를 입력으로 받아서 감시코드가 추가된 변형된 코드를 생성하는 모듈들을 보인 것이다. "PI Source Code Generator"는 원시프로그램을 분석하여 중간코드를 생성한다. 중간코드에는 스레드의 생성 및 합류, 임계구역의 시작과 끝, 공유변수에 대한 읽기/쓰기 접근사건, 그리고 내부적 비결정성 접근사건에 대한 의사코드(Pseudo Code)가 포함되어 있다. "Pseudo Call Checker"는 소스코드를 한 라인씩 읽어와 의사코드 여부를 검사한다. 의사코드가 발견되면 "Actual Call Generator"가 실제 경합탐지 엔진들을 호출할 수 있는 라이브러리 형태로 변환한다. 여기서 경합탐지 엔진들은 스레드의 생성과 합류지점에는 레이블링 엔진, 공유변수에 대한 읽기/쓰기 접근사건 및 비결정적 접근사건 지점에는 탐지프로토콜 엔진, 그리고 임계구역의 시작과 끝 지점에는 록커버 엔진이다.

이러한 감시코드를 삽입할 때 정적 분석기법을 통하여 분석한 비결정적 접근사건의 위치정보를 이용하여 비결정적 접근사건에 대한 감시코드를 임계구역의 수행순서와 관계없이 항상 동일하게 수행하도록 삽입함으로써 수행 중 경합 탐지 프로토콜에 의하여 경합의 검증이 가능하도록 한다. [그림 5]은 OpenMP 병렬 프로그램[2]에 레이블링, 탐지프로토콜, 록커버 엔진이 추가된 것을 보인 것이다. 3번 줄에 있는 "**NR_Fork**"은 레이블링 엔진이고, 7번 줄에 있는 "**NR_Add_Lock**"는 록커버 엔진이다. 그리고 8번, 9번 줄에 있는 "**CSR_Checker**"는 읽기 접근사건 탐지프로토콜 엔진이고 17번 줄에 있는 "**D_W_Checker**"는 비결정적 접근사건 탐지프로토콜 엔진이다. "**D_W_Checker**"가 17번 줄에 삽입되는 이유는 15번 줄이 수행되지 않더라도 접근사건을 감시하여 잠재적 경합을 탐지하기 위해서이다.




[그림 6] 제안된 도구의 전체모듈 구조

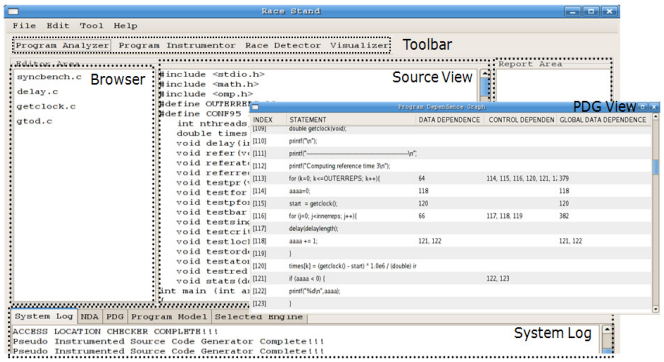
## 4. 도구의 구현

본 절에서는 제안된 도구의 전체구조와 구현환경에 대해서 기술하고, 비결정적 접근사건의 위치와 경합검증여부를 실험한 결과에 대해서 설명한다.

### 4.1 전체구조

[그림 6]은 제안된 도구의 전체 구조를 나타낸 것이다. OpenMP 병렬프로그램과 같은 소스코드가 주어지게 되면 Program Analyzer에 의해 비결정적 접근사건에 대한 존재여부를 파악하게 되고. 파악된 정보를 이용하여 변형된 중간코드가 생성된다. 중간 코드가 삽입된 코드는 다시 Program Instrumentor에 의해 실제 경합 탐지엔진을 호출하기 위한 감시코드로 대체되어 컴파일 된다. 컴파일 된 변형된 오브젝트 코드는 Race Analyzer에 의해 수행중에 경합을 탐지한다. 3절에서 설명한 [그림 2]가 Program Anzlyer이고, [그림 4]가 Program Instrumentor이다.

제안된 도구는 Java Development Kit 1.5버전의 Java 언어로 구현하였으며, 원시코드의 변수를 효과적으로 수집하기 위해서 Java Compiler Compiler 4.0으로 구현하였으면 Eclipse환경에서 개발하였다. [그림 7]은 제안된 도구의 개발된 인터페이스 화면이다. Program Analyzer, Program Instrumemtor, 그리고 Race Detector를 실행하기 위한 Toobar와 소스코드를 보기 위한 창이 있으며, 경합을 보고하기 위한 창과 시스템 로그를 보여주는 창 있다. 인터페이스 화면 위에 실행된 창은




[그림 7] 구현된 도구의 인터페이스

[표 1] 벤치마크프로그램과 합성프로그램에 대한 비결정적 접근사건 실험결과

| Program | | Shared Variables | Dependence Nodes | | | Non-deterministic Accesses | |
|---|---|---|---|---|---|---|---|
| | | | Global | Local | Control | I-VALUE | I-OCCUR |
| 벤치마크 프로그램 | Arraybench | 34 | 6 | 37 | 14 | 0 | 0 |
| | Schedbench | 26 | 11 | 45 | 19 | 0 | 0 |
| | Syncbench | 52 | 16 | 67 | 26 | 0 | 0 |
| | Jacobi | 24 | 1 | 3 | 2 | 0 | 0 |
| 합성 프로그램 | MD | 4 | 0 | 6 | 2 | 0 | 0 |
| | NNDA | 2 | 0 | 8 | 4 | 5 | 0 |
| | NNNA | 2 | 0 | 7 | 4 | 8 | 2 |
| | NEDA | 2 | 0 | 8 | 4 | 5 | 0 |
| | NENA | 2 | 0 | 9 | 6 | 7 | 2 |

PDG View 창으로써 Data Dependency, Control Dependency, 그리고 Global Data Dependency가 자동적으로 생성된 것을 보인 것이다.

### 4.2 실험결과

본 도구의 구현된 모듈들을 테스트 위해서 비결정적 접근사건들이 포함된 합성 프로그램으로 실험하고, 도구의 구현 검증을 위해서 벤치마크 프로그램인 EPCC의 Microbenchmark로 실험한다. 실험한 결과 비결정적 접근사건들이 포함된 합성 프로그램에서는 경합이 탐지되고, 벤치마크 프로그램에서는 비결정적 접근사건들이 포함된 프로그램도 없고, 그것으로 인한 경합도 발생하지 않았다.

[표 1]은 4.1절에 구현된 도구를 이용하여 벤치마크 프로그램과 합성프로그램에 대한 비결정적 접근사건에 대한 실험결과표이다. Shared Variables은 각 프로그램에서 공유변수가 될 가능성을 있는 변수의 수를 나타낸 것이며, Dependence Nodes는 Data Dependence나 Control Dependence를 가진 노드의 수를 의미한다. Data Dependence의 경우는 Global 변수에 의한 Data Dependence와 Local 변수에 의한 Data Dependence를 구별하였다. Non-Deterministic Accesses는 비결정적인 값을 가지는 노드와 비결정적으로 발생할 수 있는 노드의 수를 의미한다. 분석결과에서 보듯이 벤치마크 프로그램에서는 Non-deterministic Accesses를 볼 수 없었으며, 실험을 위해 작성된 Synthetic Program에서는 비결정적 접근사건이 정확히 분석됨을 알 수 있었다.

## 5. 결론 및 향후 과제

논문에서 제안하는 도구는 비결정적 접근사건의 위치를 찾기 위하여 프로그램의 제어흐름과 자료흐름 정보를 가진 Program Dependence Graph와 슬라이싱 알고리즘을 사용하여 정적으로 분석한다. 그리고 분석된 비결정적 접근사건의 위치 정보를 이용하여 수행 중 경합 탐지엔진으로 임계구역의 수행 양상과 관계없이 감시 되도록 감시코드를 삽입하여 경합을 탐지함으로써 잠재적으로 존재할 수 있는 경합까지 탐지하였다. 제안된 도구는 현재 OpenMP 디렉티브 기반의 프로그램에만 적용할 수 있지만 향후에 런타임 라이브러리 기반의 프로그램에서도 잠재적 경합을 탐지할 수 있도록 할 것이다.

## 참고문헌

[1] Cheng, G. I., M. Feng, C. E. Leiserson, K. H. Randall, A. F. Stark "Detecting Data Races in Cilk Programs that Use Locks," *Proc. of the Tenth Annual ACM Symposium on Parallel Algorithms and Architectures* (SPAA), pp. 298-309, ACM, June 1998.

[2] [DaMe98] Dagum, L., and R. Menon, "OpenMP: An Industry-Standard API for Shared-Memory Programming," *IEEE Computational Science & Engineering*, pp. 5(1): 46-55, IEEE, January/March 1998.

[3] Dinning, A., E. Schonberg, "An Empirical Comparison of Monitoring Algorithms for Access Anomaly Detection," *2nd Symp. On Principles and Practice of Parallel Programming*, pp. 1-10, ACM, March 1990.

[4] Dinning, A., E. Schonberg, "Detecting Access Anomalies in Programs with Critical Sections," *2nd Workshop on Parallel and Distributed Debugging*, pp. 85-96, ACM, May 1991.

[5] Ferrante, J., T. J. Watson, J. Ottenstein, J. D. Warren, "The Program Dependence Graph and Its Use in Optimization," *ACM Transactions on Programming Languages and Systems* (TOPLAS), pp. 319-349, ACM, July 1987.

[6] Jun, Y., and K. Koh, " On-the-fly Detection of Access Anomalies in Nested Parallel Loops," *Proc. of the 3rd ACM/ONR Workshop on Parallel and Distributed Debugging*, pp. 107-117, ACM, San Diego, California, May 1993. Also in SIGPLAN Notices, 28(12): 107-117, ACM, Nov. 1993.

[7] Netzer, R. H. B., B. P. Miller, "What Are Race Conditions? Some Issues and Formalizations," *ACM Letters on Programming Language and Systems,* pp. 74-88, ACM, March 1992.

[8] [OcCh03] O'Callahan, R. and J. Choi, "Hybrid Dynamic Data Race Detection," *Proc. of ACM SIGPALN Symp. on Principle and Practice of Parallel Programming* (PPoPP), San Diego, California, ACM, June 2003.

[9] Praun, C., T. R. Gross, "Object Race Detection," *Proc. of the 16th ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages, and Applications*, pp. 70-82, ACM, October 2001.

[10] Stefan S., M. Burrows, G. Nelson, P. Sobalvarro, T. Anderson, "Eraser: a dynamic data race detector for multithreaded programs," *ACM Transactions on Computer Systems* (TOCS), pp. 391-411, ACM, Nov. 1997.

[11] Yu, Y., T. Rodeheffer, W. Chen, "Race Track: Efficient Detection of Data Race Conditions via Adaptive Tracking," *Proc. of the twentieth ACM Symposium on Operating Systems Principles*, pp. 221-234, ACM, Oct. 2005.