내포병렬성을 가진 공유메모리 프로그램의 수행중 최초경합 탐지를 위한 효율적 기법

(Efficient On-the-fly Detection of First Races in Shared-Memory Programs with Nested Parallelism)

하 금 숙 † 전 용 기 †† 유 기 영 †††
(Keum-Sook Ha) (Yong-Kee Jun) (Kee-Young Yoo)

요 약 내포병렬성을 가진 공유메모리 병렬프로그램의 효과적인 디버깅을 위해서, 프로그램의 비결정적 수행을 최초로 초래하는 경합을 효율적으로 탐지하는 것이 중요하다. 이러한 최초경합을 수행 중에 탐지하는 기존의 기법은 두 번의 프로그램 수행을 통해서 탐지하면서 각 공유변수마다 프로그램의 최대병렬성에 의존적인 크기의 접근역사를 유지하므로 비효율적인 수행시간과 기억공간을 요구한다. 본 논문에서는 두 번의 프로그램 수행을 통해서 수행 중에 각 공유변수에 대한 접근역사를 상수적 크기로 유지하므로, 각 접근사건의 수행 시에 상수적 복잡도의 사건비교 횟수와 기억 공간만을 요구하는 새로운 최초경합 탐지기법을 제안한다. 그러므로 본 기법은 내포병렬성을 가진 공유메모리 병렬프로그램의 디버깅을 위해서 보다 효율적이고 실용적인 경합탐지를 가능하게 한다.

키워드 : 수행중 최초경합 탐지, 내포병렬성, 효율성, 상수적 복잡도

Abstract For debugging effectively the shared-memory programs with nested parallelism, it is important to detect efficiently the first races which incur non-deterministic executions of the programs. Previous on-the-fly technique detects the first races in two passes, and shows inefficiencies both in execution time and memory space because the size of an access history for each shared variable depends on the maximum parallelism of program. This paper proposes a new on-the-fly technique to detect the first races in two passes, which is constant in both the number of event comparisons and the space complexity on each access to shared variable because the size of an access history for each shared variable is a small constant. This technique therefore makes on-the-fly race detection more efficient and practical for debugging shared memory programs with nested parallelism.

Key words : on-the-fly detection of first races, nested parallelism, efficiency, constant complexity

1. 서 론

공유메모리 병렬프로그램의 수행에서 병행 스레드들이 동일한 공유 변수를 사용하고, 적절한 동기화 없이 적어도 하나의 쓰기 접근을 포함하면 경합(race)[1]이라는 접근 오류가 발생한다. 경합은 프로그래머의 의도와는 달리 프로그램을 비결정적(nondeterministic)으로 수행하게 하므로 병렬프로그램에서 발생할 수 있는 오류들을 효과적으로 디버깅하기 위하여서는 반드시 탐지되어야 한다. 특히 경합들 중에서 가장 먼저 발생하여 다른 경합에 영향을 받지 않은 최초경합(first race)[1, 2, 3]을 효율적으로 탐지하는 것이 중요하다. 그 이유는 최초경합을 제거하면 이에 영향을 받은 다른 경합들이 나타나지 않을 수도 있기 때문이다.

병렬프로그램의 최초경합을 수행 중에 탐지하는 기존의 기법[2, 4, 5, 6, 7] 중에서 내포병렬성을 위해 가장 효율적인 기법[5]은 두 번의 프로그램 수행을 필요로 한다. 첫 번째 수행에서는 각 공유 변수의 접근사건들을 감시하면서 공유 자료구조인 접근역사(access history) 내의 접근들과 비교하고 유지함으로써 최초경합을 발생

† 종신회원 : 구미1대학 컴퓨터전공 교수
ksha@kumi.ac.kr
†† 종신회원 : 경상대학교 컴퓨터과학과 교수
(컴퓨터·정보통신연구소 연구원)
jun@nonage.gsnu.ac.kr
††† 종신회원 : 경북대학교 컴퓨터공학과 교수
yook@knu.ac.kr
논문접수 : 2003년 3월 5일
심사완료 : 2003년 3월 31일

시킬 수 있는 후보사건(candidate event)[3, 4, 5, 7]들의 부분집합을 수집한다. 그리고 두 번째 수행에서는 각 접근사건들에 대하여 이미 수집된 후보사건들과의 병행성 관계를 조사하여 후보사건들의 집합을 만들고 이들을 최초경합으로 보고한다. 그러나 이러한 기법은 후보사건들의 집합을 수집하기 위해서 각 공유 변수 마다 프로그램의 최대병렬성에 의존하는 크기의 접근역사를 유지해야 하므로 각 접근 사건의 수행 시에 비효율적인 수행 시간과 기억 공간을 요구한다.

본 논문에서는 내포병렬성을 가진 병렬 프로그램의 최초경합을 탐지하기 위해서 프로그램을 두 번 수행하여 감시할 때, 기존의 기법과 달리 각 공유변수에 대한 접근 역사를 상수적 크기로 유지하여, 각 접근 사건의 수행 시에 상수적 복잡도의 수행 시간과 기억 공간만을 요구하는 새로운 수행중 탐지 기법을 제안한다. 이 기법의 첫 번째 수행에서는 각 공유 변수의 접근 사건들을 선행(happened-before) 관계[9]와 좌우(left-of) 관계[10]로써 비교하여 접근역사 내의 접근들을 상수개로 유지하고, 접근역사 내의 사건들과 처음으로 경합을 구성하는 사건들을 최초경합을 발생시키는 후보사건으로 수집한다. 두 번째 수행에서는 각 접근 사건들을 감시하면서 첫 번째 수행에서 수집된 후보사건들과의 선행 관계와 좌우 관계를 검사하여 최초경합을 구성하는 후보사건들의 집합을 완성하여 최초경합으로 보고한다.

이러한 수행중 경합탐지 기법의 효율성은 두 가지 측면으로 분석할 수 있다. 첫째는 수행 중에 접근 사건들 간의 논리적 순서를 판단하기 위한 병행성 정보가 필요하므로, 각 스레드의 생성 및 종료 시에 이러한 정보를 생성하기 위한 시간 및 공간이 필요하다. 둘째는 각 접근 사건이 경합에 포함되는지를 판단하기 위하여 이전에 발생된 사건들의 병행성 정보를 저장하는 접근역사가 필요하므로, 각 접근 사건의 수행 시에 이러한 구조를 유지하기 위한 시간 및 공간이 필요하다. 본 논문에서 제안된 기법은 내포병렬성을 가진 공유메모리 병렬 프로그램에서 각 공유 변수에 대한 접근역사의 항목 수를 상수개로 제한하고, 각 항목에 저장되는 병행성 정보도 NR Labeling 기법[8, 15]을 적용하여 상수적 복잡도를 가지게 한다. 기존의 기법과는 달리, 이 기법은 각 접근 사건의 수행 시에 프로그램의 최대병렬성에 무관하게 상수적 복잡도의 사건비교 횟수와 기억공간만으로 수행중 최초경합 탐지를 보장하므로, 내포병렬성을 가진 공유메모리 병렬프로그램의 디버깅을 위해서 효율적이고 실용적인 경합탐지를 가능하게 한다.

본 논문의 2장에서는 병렬프로그램에서의 최초경합 개념을 소개하고, 최초경합 탐지를 위한 기존 기법들의 문제점을 효율성 측면에서 제기한다. 3장에서는 내포병렬성이 있는 프로그램을 위해서 각 접근 사건의 수행 시에 상수적인 시간 및 공간 복잡도로 수행되는 효율적인 탐지 프로토콜을 제시한다. 제시된 프로토콜을 분석하기 위해서, 4장에서는 본 프로토콜 알고리즘의 정당성을 입증하기 위한 주요 정리들을 증명하고, 알고리즘의 효율성을 시간 및 공간 복잡도의 측면에서 분석한다. 마지막으로, 5장에서는 본 연구의 결론 및 향후 과제를 제시한다.

2. 연구 배경

본 장에서는 본 연구의 대상이 되는 내포병렬성을 가진 병렬프로그램 모델을 소개하고, 내포병렬성이 있는 병렬프로그램에서의 최초경합 개념을 소개한 후, 기존의 최초경합 탐지 기법들의 문제점을 제기한다.

2.1 병렬프로그램 모델과 최초경합

본 논문에서는 산업표준 병렬 프로그램 모델로서 루프 수준의 내포병렬성을 지원하는 OpenMP[12, 13] 병렬프로그램 모델을 고려한다. OpenMP 프로그램의 수행에서 'C$OMP PARALLEL DO' 디렉티브는 병렬 스레드를 생성(fork)을 지시하고 'C$OMP END PARALLEL DO' 디렉티브는 병렬 스레드들의 합류(join)을 지시한다. 대상으로 하는 내포병렬성은 병행하는 스레드들 간에 동기화 명령이 없는 것으로 가정한다. 하나의 스레드 내에 또 다른 하나의 루프가 나타나면 이 루프는 내포되었다(nested)고 한다. 각각의 병렬 루프가 가질 수 있는 내포 수준(nesting level)은 외부 루프의 내포 수준보다 1이 증가한 값이며, 임의 병렬 루프에서 나타나는 최대 내포수준을 내포 깊이(nesting depth)라고 한다. 그림 1은 내포 깊이가 2인 병렬 루프를 가진 OpenMP Fortran 프로그램이다. 'C$OMP SHARED(X)'는 변수 X를 공유 변수로 지정하고, 'C$OMP PRIVATE(I, J1, J2)'는 변수 I, J1 그리고 J2를 각 스레드에서 사용하는 지역 변수로 선언하는 디렉티브이다.

병렬프로그램의 수행 시의 스레드들 간의 병행성 관계는 POEG(Partial Order Execution Graph)[14]라는 방향성 있는 비순환 그래프로 나타낼 수 있다. POEG는 병렬프로그램의 부분적인 수행 순서관계를 표현한다. POEG에서 정점(vertex)은 병렬 스레드의 생성 및 합류 명령과 임의 공유 변수 X에 대한 접근 사건을 표현하며, 임의의 정점에서 시작하는 간선(arc)은 그 정점으로부터 생성된 스레드를 나타낸다. 그림 2는 그림 1의 병렬프로그램을 POEG로 표현한 것이다. 여기에서 r과 w

```
C$OMP PARALLEL DO SHARED(X)
C$OMP PRIVATE(I, J1, J2)
      DO I = 1, 2
       . . . = X                          {r0, r8}
      IF (I .EQ. 2) THEN
C$OMP PARALLEL DO
      DO J1 = 1, 2
      . . . = X                           {r1, r2}
      . . . = X                           {r3, r4}
C$OMP END PARALLEL DO
      END IF
      . . . = X                           {r5, r9}
      IF (I .EQ. 2) THEN
C$OMP PARALLEL DO
      DO J2 = 1, 2
      IF . . . THEN . . . = X             {r6}
      IF . . . THEN X = . . .             {w10}
      X = . .                             {w7, w13}
C$OMP END PARALLEL DO
      END IF
      X = . . .                           {w11, w14}
      IF (I .EQ. 1) THEN . . . = X        {r12}
C$OMP END PARALLEL DO
```

그림 1 OpenMP 병렬프로그램

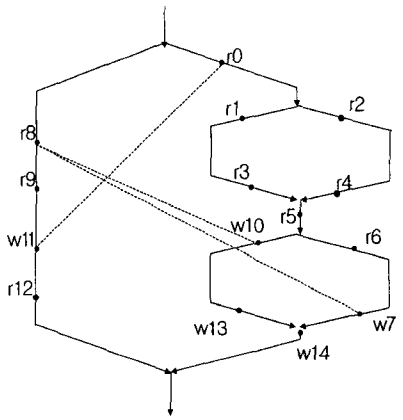

그림 2 POEG

는 해당 스레드에서 발생하는 읽기와 쓰기 사건들을 표현하고 이 기호와 함께 표기된 숫자는 임의의 수행에서 나타날 수 있는 접근 사건들의 발생 순서를 의미한다.

임의의 사건 e_i와 e_j를 포함하는 POEG에서 사건 e_i에서 e_j로 가는 경로가 존재하면 e_i는 e_j에 선행(happend-before)하거나 순서화(ordered)되었다고 하고 '$e_i \rightarrow e_j$'로 나타낸다. 본 논문에서는 e_i나 e_j 중에서 하나의 사건만 존재하는 경우에도 스스로 순서화된 것으로 한다. e_i와 e_j 사이에 경로가 없으면 e_i와 e_j는 '병행(concurrent)하다'라고 하고 '$e_i \parallel e_j$'로 나타낸다. 하나의 공유 변수에 대한 두 사건들 중에서 적어도 하나가 쓰기 사건이면 이들은 '대립하는(conflicting) 사건'이라 한다. e_i와 e_j가 대립사건이고 병행하면 두 사건은 경합(race)을 구성하는데 이를 'e_i-e_j'로 나타낸다. 주어진 두 사건 e_i와 e_j에서 e_i가 e_j에 선행하고 e_i가 경합에 포함되면 e_j는 e_i에 의해 '영향받은(affected) 사건'이라 한다. 임의 경합을 구성하는 두 사건 e_i와 e_j 중에서 어느 것도 임의의 다른 사건에 의해 영향받은 사건이 아니면 e_i-e_j는 '영향받지 않은(unaffected) 경합'이라 한다. 경합을 구성하는 사건들 e_i와 e_j 중에서 하나만 다른 사건에 의해 영향을 받으면 '부분적으로 영향받은(partially affected) 경합'이라고 한다. 이 때, 서로 부분적으로 영향받은 경합들이 두 개 이상인 임의 경합집합을 '얽힘(tangle)'이라 하고 얽힘에 속한 임의의 경합을 '얽힌(tangled) 경합'이라고 한다. 또한 모든 접근 사건들이 기껏해야 하나의 얽힌 경합에 영향받은 경합들의 얽힘을 '최초얽힘(first tangle)'이라고 한다. 이러한 영향받지 않은 경합이거나 그렇지 않으면 최초얽힘에 속한 경합을 '최초경합(first race)'이라고 한다.

그림 2에는 23개의 경합이 나타날 수 있는데 이들 중 {*r0-w11, w7-r8, r8-w10*} 경합만이 최초얽힘에 속한 경합인 최초경합이다. 이러한 최초경합을 탐지하여 제거하면 이에 영향받은 다른 경합들이 사라질 수 있기 때문에 프로그램의 디버깅에 있어서 중요하다.

2.2 기존의 수행중 최초경합 탐지 기법

기존의 수행중 최초경합 탐지 기법은 모든 접근 사건을 대상으로 경합을 탐지하는 기법[2, 6]과 최초경합의 가능성이 있는 후보사건만으로 경합을 탐지하는 기법[4, 5, 7]이 있다.

Race Frontier 기법[2]은 경합이 탐지된 첫 번째 지점을 cut-set으로 지정하여 이 지점까지 재수행하면서 최초경합을 탐지한다. 그러나 이 기법은 프로그래머에게 중단점(breakpoint) 설정이나 반복 수행을 통한 경합 감시 기능을 추가로 요구하므로 최초경합 탐지를 위해서

필요로 하는 반복 수행 횟수가 비결정적이다. RecPlay 기법[6]은 첫 번째 수행에서는 동기화 사건만을 추적 파일(trace file)에 기록하고, 두 번째 수행에서는 동기화 사건들 사이에 읽기와 쓰기 사건 집합을 수집하고 비교하여 경합 존재를 확인하며, 세 번째 수행에서 최초경합을 초래한 기계 명령어(machine instruction)의 위치 탐지를 시도하는 기법이다. 이 기법은 최초경합의 위치를 기계 명령어의 위치로써 보고하므로 프로그램 위치에서의 보고를 위하여서는 추가적인 보완이 필요하며, 최초경합을 탐지하기 위해서 적어도 세 번의 수행 감시를 필요로 한다.

후보사건을 수집하여 최초경합을 탐지하는 기법들 중에서 단일 내포병렬성을 위한 단일 수행 기법[4]은 기존의 공유 변수들의 접근에서 야기되는 병목 현상을 감소시켜 최초경합을 탐지하는 기법으로 시간복잡도면에서 효율성이 매우 높다. 내포병렬성과 순서적 동기화를 대상으로 하는 후보사건 수집 기법은 두 단계 기법[5]으로서, 첫 번째 수행에서 모든 접근 사건들을 감시하여 후보사건들의 부분 집합을 모으고, 두 번째 수행에서는 첫 번째 수행에서 얻어진 후보사건들 중에서 최초경합에 관련된 후보경합을 찾아내어 최초경합으로 보고한다. 그러나 이 기법은 후보사건들의 집합을 수집하기 위해서 각 공유 변수 마다 프로그램의 최대병렬성에 의존하는 크기의 접근역사를 유지해야 하므로 비효율적인 수행 시간과 기억 공간을 요구한다. 임계구역 동기화를 가진 내포병렬 프로그램을 대상으로 후보사건을 수집하는 기법[7]은 시각적으로 각 접근 사건들을 감시하고 추적하여 최초경합을 탐지하도록 한 확장적 기법이다. 그러나 이 기법은 시각적으로 경합을 나타내기 위하여 수행 중에 각 접근들의 정보를 추적하여 저장하고, 사후 단계에서 이들을 활용하여 경합을 시각화하므로 공간 효율성 면에서 부담이 크다.

2.3 수행중 최초경합 탐지 기법의 효율성

이러한 수행중 경합탐지 기법의 효율성은 두 가지 측면으로 분석할 수 있다. 첫째는 수행 중에 접근 사건들 간의 논리적 순서를 판단하기 위한 병행성 정보가 필요하므로, 각 스레드의 생성 및 종료 시에 이러한 정보를 생성하기 위한 시간 및 공간이 필요하다. 둘째는 각 접근 사건이 경합에 포함되는지를 판단하기 위하여 이전에 발생된 사건들의 병행성 정보를 저장하는 접근역사가 필요하므로, 각 접근 사건의 수행 시에 이러한 구조를 유지하기 위한 시간 및 공간이 필요하다. 그러므로 각 접근 사건의 수행 시에 요구되는 공간복잡도와 시간복잡도는 각 접근역사를 구성하는 항목의 수와 각 항목에 저장되는 병행성 정보의 크기에 의존적이다.

본 연구에서는 이러한 병행성 정보를 위해서 NR 레이블링[8, 15]을 사용하는데, 이 기법은 내포병렬성을 갖는 병렬프로그램에서 병행성 정보 생성 및 적용을 위한 효율성이 가장 우수한 기법이다. 이 기법을 적용하면, 접근역사의 각 항목에 저장되는 접근 사건 정보인 병행성 정보의 크기가 상수적이므로 각 접근역사의 공간 및 시간 복잡도는 접근역사를 구성하는 항목 수에 의존적이 된다.

내포병렬성을 갖는 병렬프로그램에서 최초경합을 탐지하기 위한 가장 효율적인 기존의 기법[5]은 수행중 최초경합 탐지 시에 각 공유 변수들에 대한 접근역사의 항목 수가 프로그램의 최대병렬성(T)에 비례한다. 그러므로 이러한 접근역사를 유지하기 위한 기억 공간과 각 접근 사건의 수행 시에 접근역사에 저장된 사건들과와 비교 회수도 최대병렬성에 의존적인 $O(T)$ 복잡도를 요구하므로 비효율적이다. 본 논문에서는 이러한 복잡도를 상수적으로 개선한 효율적인 수행중 최초경합 탐지 프로토콜을 제시한다.

3. 효율적인 최초경합 탐지 기법

본 연구는 프로그램의 수행 중에 최초경합의 가능성이 있는 후보사건을 수집하여 최초경합을 탐지하는 기법이다. 본 절에서는 후보사건을 수집하기 위하여 모든 접근 사건을 감시할 때 각 공유 변수마다 항상 상수적 크기의 접근역사를 유지하여 수행 중에 최초경합을 탐지할 수 있는 새로운 프로토콜을 제안한다. 프로토콜은 두 단계의 수행으로 구성되는데 1단계에서는 변두리 후보사건을 수집하고 2단계에서는 변두리 후보사건을 이용하여 후보사건 집합을 완성하고 최초경합을 보고한다. 본 절에서 프로토콜의 정당성을 위해서 보인 정리들에 대한 증명은 다음 절에서 보인다.

3.1 변두리 후보사건

최초경합을 구성하는 후보사건[3, 4, 5, 7]으로는 읽기(read) 후보사건, 쓰기(write) 후보사건 그리고 읽기-쓰기(read-write) 후보사건이 있다. 어떤 읽기 (혹은 쓰기) 사건에 선행하는 다른 접근 사건이 존재하지 않으면 그 사건을 읽기 혹은 쓰기 후보사건이라 한다. 어떤 쓰기 사건 w_i에 선행하는 임의의 다른 쓰기 사건이 존재하지 않고 w_i에 선행하는 읽기 후보사건 r_k가 존재한다면 w_i를 읽기-쓰기 후보사건이라고 한다. 읽기 혹은 쓰기 후보사건은 항상 스스로 유효(effective)하지만, 읽기-쓰기 후보사건은 병행하는 다른 쓰기 후보사건이 없을 경우에만 유효하다.

수행 중 임의의 공유 변수에 대한 모든 접근 사건은 접근 사건들 간의 선행 관계와 좌우 관계를 비교하여 접근 유형에 따라 접근역사라는 공유 자료구조에 유지된다. 본 기법에서의 좌우 관계는 Mellor-Crummey의 좌우 관계[10]를 재정의하여 사용하는데, 임의의 두 접근 사건 $\{e_i, e_j\}$에 대한 좌우 관계는 $(e_i \prec e_j)$ 혹은 Leftof(e_i, e_j)로 표현한다.

[정의 3.1] 동일한 유형의 두 접근 사건 $\{e_i, e_j\}$가 존재하고, Leftof(e_h, e_i) ∧ Leftof(e_j, e_k)를 만족하는 $\{e_h, e_k\}$가 존재하지 않으면, $\{e_i, e_j\}$는 외부에(outside) 존재한다고 한다. 그렇지 않으면 $\{e_i, e_j\}$는 $\{e_h, e_k\}$보다 내부에(inside) 존재한다고 한다.

예를 들어, 그림 2에서 두 접근 사건 $\{r8, r6\}$은 Leftof$(r_x, r8)$ ∧ Leftof$(r6, r_y)$를 만족하는 $\{r_x, r_y\}$가 존재하지 않으므로 $\{r8, r6\}$는 외부에 존재하는 사건이다. 두 접근 사건 $\{r1, r3\}$은 Leftof$(r_x, r1)$ ∧ Leftof$(r3, r_y)$ 혹은 Leftof$(r_x, r3)$ ∧ Leftof$(r1, r_y)$를 만족하는 $\{r_x, r_y\}$가 존재하므로 $\{r1, r3\}$는 내부에 존재하는 사건이다.

본 기법에서는 임의의 공유 변수 X에 대한 접근역사 AH(X)가 두 개의 부분집합 {AH(X, R), AH(X, W)}로 구성된다. AH(X, R)은 읽기 접근역사로서, 가장 최근에 발생한 읽기 접근 사건들의 좌우 관계를 고려하여 [정의 3.1]을 만족하는 두 외부에 존재하는 사건들을 저장하고 {AH(X, R_L), AH(X, R_R)}으로 표현한다. AH(X, W)는 쓰기 접근역사로 가장 최근의 쓰기 접근 사건이다. 그러므로 접근역사의 크기는 항상 상수개로 유지된다.

후보사건을 탐지하기 위하여서는 임의 공유 변수에 대한 접근 사건이 발생할 때마다 후보사건 결정이 수행되어야 한다. 본 기법에서는 첫 번째 수행 단계에서 공유 변수에 대한 임의의 접근 사건이 접근역사 내의 접근 사건들과 병행하고 대립되는 사건인가를 고려하여 경합 존재 여부를 판단한다. 그리고 그 경합에 포함된 접근 사건이 접근역사에 존재하는 사건들에 병행하면서도 보다 더 외부에 존재하는 변두리 사건인가를 판단하여 이 변두리 사건(frontier event)이 후보사건인지 여부를 결정한다.

[정의 3.2] 임의의 접근 사건이 존재할 때, 이 사건이 경합에 포함되고 외부에 존재하면, 그 사건을 변두리 사건(frontier event)이라 한다.

예를 들어, 그림 2에서 $r8$은 경합 $\{w7\text{-}r8, r8\text{-}w10\}$에 포함되면서 외부에 존재하는 사건이므로, $r8$은 변두리 사건이다. 변두리 후보사건의 수는 기껏해야 두 개이므로, 본 기법에서는 이 사건들을 저장하는 자료구조인 후보역사(candiadate history) CH(X) 또한 두 개의 부분집합 {CH(X, R), CH(X, W)}로 구성된다. CH(X, R)에는 좌우 관계를 고려한 두 개의 읽기 변두리 사건이 {CH(X, R_L), CH(X, R_R)}에 각각 저장되고, CH(X, W)에는 CH(X, R)과 마찬가지로 좌우 관계를 고려한 두 개의 쓰기 변두리 사건이 {CH(X, W_L), CH(X, W_R)}에 저장된다.

그리고 두 번째 단계에서는 이러한 변두리 후보사건과 경합인 접근 사건들 중에서 임의의 공유 변수 X에 대한 후보사건 집합(candidate set)을 보고한다. 후보사건 집합 CS(X)는 유효한 후보사건인 두 개의 부분집합 {CS(X, R), CS(X, W)}로 구성된다. CS(X, R)은 읽기 후보사건 집합이고, CS(X, W)는 쓰기 후보사건 집합이다.

3.2 1단계 탐지 프로토콜

매 접근 사건에 대하여 변두리 후보사건인가를 판단하여 저장하기 위하여 1단계 수행에서는 접근사건의 유형에 따라 checkread_1st() 혹은 checkwrite_1st() 함수가 수행된다. 그림 3에서 보인 1단계 탐지 프로토콜에서 current는 공유 변수 X에 대한 현재의 접근 사건을 나타낸다. current가 읽기 접근 사건인 경우를 보자. checkread_1st()가 수행되어 1번 줄에서 쓰기 접근역사 내의 사건과 선행 관계를 검사하여 경합 발생 여부를 판단한다. 3-6번 줄에서는 current와 읽기 접근역사 내의 사건들과의 좌우 관계를 검사하여 접근역사의 내용을 갱신하고, 1번 줄에서 경합이 발생하지 않았으면 변두리 후보사건이 아니므로 반환하고, 경합이 발생하였으면 10-14번 줄에서 기존 후보역사 내의 사건들과 좌우 관계를 고려하여 변두리 후보사건이면 저장한다.

다음으로 current가 쓰기 접근 사건인 경우를 보자. checkwrite_1st()가 수행되어 1-2번 줄에서 접근역사 내의 모든 사건들과의 선행 관계를 검사하여 경합 발생 여부를 판단한다. 4번 줄에서는 current를 쓰기 접근역사로 갱신하고, 1-2번 줄에서 경합이 발생하지 않았으면 변두리 후보사건이 아니므로 반환한다. 경합이 발생하였으면 7-12번 줄에서 기존 후보역사 내의 사건들과 좌우 관계를 고려하여 변두리 후보사건이면 저장한다. 앞 절에서 언급했듯이 본 논문에서는 후보 집합으로 읽기와 쓰기 후보사건만을 수집하여 저장하므로, 읽기-쓰기 후보사건은 간접적으로 판단되어진다. 읽기-쓰기 후보사건이 존재하는가의 여부를 판단하기 위해서, 수집된 모든 변두리 후보사건을 13번 줄에서 CS$(X, Backup)$으로 저장하고, 2단계 프로토콜의 수행 결과와 비교하여 그 여부를 결정한다. 일반적으로 쓰기 후보사건이 발생

```
0: checkread_1st(X, current)
1: if ¬ Ordered(AH(X, W), current) then
2:         racing_current := true;
3: if ¬ Leftof(AH(X, RL), current) then
4:         AH(X, RL) := current;
5: if ¬ Leftof(current, AH(X, RR)) then
6:         AH(X, RR) := current;
7: if ¬ racing_current then
8:         return;
9: if CH(X, RL) = ∅ or
10:    Leftof(current, CH(X, RL)) then
11:        CH(X, RL) := current;
12: if CH(X, RR) = ∅ or
13:    Leftof(CH(X, RR), current)) then
14:        CH(X, RR) := current;
15: end checkread_1st

0: checkwrite_1st(X, current)
1: for all c in AH(X) do
2:     if ¬ Ordered(c, current) then
3:         racing_current := true;
4: AH(X, W) := current;
5: if ¬ racing_current then
6:         return;
7: if CH(X, WL) = ∅ or
8:    Leftof(current, CH(X, WL)) then
9:         CH(X, WL) := current;
10: if CH(X, WR) = ∅ or
11:    Leftof(CH(X, WR), current)) then
12:        CH(X, WR) := current;
13: add current to CS(X, Backup);
14: halt;
15: end checkwrite_1st
```

그림 3 1단계 탐지 프로토콜

한 스레드에서의 쓰기 후보사건이 선행하는 모든 접근 사건들은 후보사건의 정의에 의해서 후보사건이 아니므로 더 이상 수행하지 않고 중지(halt)시킨다. 이렇게 함으로써 불필요한 수행 감시 시간을 제거한다.

표 1은 그림 2의 매 접근 사건이 발생할 때 최초경합

표 1 1단계 탐지 프로토콜의 예

접근 사건	1단계 수행							
	AH			CH				CS
	R_L	R_R	W	R_L	R_R	W_L	W_R	backup
r0	r0	r0						
r1	r1	r1						
r2	r1	r2						
r3	r3	r2						
r4	r3	r4						
r5	r5	r5						
r6	r6	r6						
w7	r6	r6	w7					
r8	r8	r6	w7	r8	r8			
r9	r9	r6	w7	r8	r8			
w10	r9	r6	w10	r8	r8	w10	w10	w10
w11	r9	r6	w11	r8	r8	w11	w10	w10, w11
r12	HALTED!							
w13	HALTED!							
w14	HALTED!							

탐지 1단계 프로토콜이 수행되고 있는 과정을 접근역사와 후보역사의 내용으로 보이고 있다. 첫 접근 사건으로 *r0*가 AH(X, R_L)과 AH(X, R_R)에 추가되고, 연이어 *r1*이 발생하면 좌우 관계를 고려하여 AH(X, R_L)과 AH(X, R_R)에 *r0*가 제거되고 *r1*이 저장된다. *r2*가 발생하면 좌우관계로 인하여 AH(X, R_R)에 *r1*이 제거되고 *r2*가 저장된다. 이런 형태로 계속해서 *r6*가 발생할 때 까지 AH(X, R_L)과 AH(X, R_R)에서 *r5*가 제거되고 *r6*이 저장된다. 이는 *r6* 발생 후에 수행되면서 *r0*에서 *r5* 까지와 경합을 이루는 어떤 접근 사건도 *r6*와 경합을 이루기 때문이다. 이런 방법으로 수행하면 1단계 수행 결과로 읽기 접근역사에는 *r9*와 *r6*이 최근에 발생한 외부에 존재하는 읽기 사건으로 저장되고, 쓰기 접근역사에는 최근에 발생한 쓰기 접근 사건 *w11*이 저장되어 있다. 쓰기 접근 사건 *w7*, *w10*, 그리고 *w11*이 수행한 스레드에서 이들 사건들이 선행한 접근 사건들은 모두 수행이 중지된다. 또한 후보역사에는 *r8*, *w10* 그리고 *w11*이 각각 읽기 및 쓰기 변두리 후보사건으로 수집되어 있다. 또한 쓰기 후보사건 *w10*과 *w11*은 읽기-쓰기 후보사건을 간접적으로 판단하기 위하여 CS(*X*, *Backup*)에 저장된다. 1단계에서만 유지되어지는 접근역사 내의 접근들은 항상 두 개의 읽기 사건과 한 개의 쓰기 사건으로 구성된다. 후보역사 내의 후보사건들도 변두리 사건들로

두 개의 읽기 변두리 후보사건과 두 개의 쓰기 변두리 후보사건이다.

1단계 탐지 프로토콜의 의미를 정리해 보면 다음과 같다. 첫째, (checkread_1st()의 1-6번 줄과 checkwrite_1st()의 1-4번 줄) 임의의 수행에서 경합이 존재하면, 적어도 하나의 쓰기 사건이나 외부에 존재하는 읽기 사건이 경합에 포함된다. 그리고 동일한 접근 유형의 두 접근 사건 $\{e_i, e_j\}$가 $(e_i \rightarrow e_j)$를 만족하고, $(e_i \parallel e_k) \wedge (e_j \parallel e_k)$를 만족하는 대립사건 e_k가 존재할 때, e_i와 e_k가 경합이면, e_j와 e_k도 경합이다. 둘째, (checkread_1st()의 2번 줄과 7-8번 줄, 그리고 checkwrite_1st()의 3번 줄과 5-6번 줄) 임의의 접근 사건이 존재할 때, 이 사건과 경합인 접근 사건이 존재할 필요충분조건은 이 사건이 경합에 포함되는 것이다. 셋째, (checkread_1st()의 9-14번 줄과 checkwrite_1st()의 7-12번 줄) 경합에 포함된 임의의 읽기나 쓰기 사건이 존재할 때, 이 사건이 변두리 사건이고 이 사건에 선행하는 다른 읽기나 쓰기 변두리 사건이 존재하지 않으면, 이 사건은 변두리 후보사건이다. 넷째, (checkwrite_1st()의 14번 줄) 경합에 포함된 임의의 쓰기 사건이 존재할 때, 이 사건이 선행하는 다른 쓰기 사건이 존재하지 않는다.

[정리 3.1] 임의의 수행에서 경합이 존재하면, 적어도 하나의 쓰기 사건이나 외부에 존재하는 읽기 사건 e_j가 경합에 포함되고, e_j와 경합에 포함되는 접근 사건 e_k와 경합에 포함되면서 e_j에 선행하는 접근 사건 e_i의 집합이 존재한다.

[정리 3.2] 경합에 포함된 임의의 접근 사건이 존재할 때, 이 사건이 변두리 사건이고 이 사건에 선행하는 다른 동일한 접근 유형의 변두리 사건이 존재하지 않으면, 이 사건은 변두리 후보사건이다.

[정리 3.3] 경합에 포함된 임의의 쓰기 사건이 존재할 때, 이 사건에 선행하며 경합에 포함된 어떤 쓰기 사건도 존재하지 않으면, 이 쓰기 사건은 후보사건이다.

3.3 2단계 탐지 프로토콜

2단계에서는 매 접근 사건들을 감시하면서 1단계 수행에서 수집한 후보사건들과의 선행 관계와 좌우 관계를 검사하여 후보사건 집합을 완성하여 최초경합을 보고한다. 그림 4에서 보인 2단계 탐지 프로토콜에서 current가 읽기 접근 사건이면 checkread_2nd()가 수행된다. 1번 줄에서 후보역사 내의 쓰기 변두리 후보사건과의 경합 발생 여부를 판단한다. 경합이 발생하지 않으면 후보사건이 아니므로 반환한다. 경합이 발생하면6 11줄에서 읽기 후보역사 내의 사건들과 좌우 관계를

```
0:  checkread_2nd(X, current)
1:  for all c in CH(X, W) do
2:      if ¬ Ordered(c, current) then
3:          racing_current := true;
4:  if ¬ racing_current then
5:          return;
6:  if CH(X, R_L) = ∅ or
7:    Leftof(current, CH(X, R_L)) then
8:          CH(X, R_L) := current;
9:  if CH(X, R_R) = ∅ or
10:   Leftof(CH(X, R_R), current) then
11:         CH(X, R_R) := current;
12: add current to CS(X, R);
13: halt;
14: end checkread_2nd

0:  checkwrite_2nd(X, current)
1:  for all c in CH(X) do
2:      if ¬ Ordered(c, current) then
3:          racing_current := true;
4:  if ¬ racing_current then
5:          return;
6:  if CH(X, W_L) = ∅ or
7:    Leftof(current, CH(X, W_L)) then
8:          CH(X, W_L) := current;
9:  if CH(X, W_R) = ∅ or
10:   Leftof(CH(X, W_R), current) then
11:         CH(X, W_R) := current;
12: add current to CS(X, W);
13: halt;
14: end checkwrite_2nd
```

그림 4 2단계 탐지 프로토콜

고려하여 경합을 발생시킨 current가 새로운 변두리 후보사건이면 후보역사에 저장한다. 13줄에서는 읽기 후보사건이 발생한 스레드의 수행을 중지시킨다. current가 쓰기 접근 사건이면 checkwrite_2nd()가 수행된다. 1번 줄에서 후보역사 내의 모든 변두리 후보사건과의 경합 발생 여부를 판단한다. 그리고 이 후의 과정은 checkread_2nd() 경우와 유사하다.

표 2는 그림 2의 매 접근 사건이 발생할 때 2단계 프로토콜이 수행되고 있는 과정을 후보역사의 변두리 후

표 2 2단계 탐지 프로토콜의 예

접근 사건	2단계 수행					
	CH				CS	
	R_L	R_R	W_L	W_R	R	W
r0	r8	r0	w11	w10	r0	
r1	HALTED!					
r2	HALTED!					
r3	HALTED!					
r4	HALTED!					
r5	HALTED!					
r6	HALTED!					
w7	HALTED!					
r8	r8	r0	w11	w10	r0, r8	
r9	HALTED!					
w10	HALTED!					
w11	HALTED!					
r12	HALTED!					
w13	HALTED!					
w14	HALTED!					

보사건과 최종적으로 수집된 후보사건들로써 보이고 있다. 첫 접근 사건으로 *r0*가 발생하면 쓰기 후보역사 내의 변두리 후보사건들과의 경합 발생 여부를 검사하고 경합이 발생하였으므로 이를 읽기 후보역사 내의 변두리 후보사건들과 좌우 관계를 고려한 뒤 읽기 후보역사의 내용을 갱신하고 이 사건을 읽기 후보사건으로 저장한다. *r0*가 후보사건으로 보고되면, *r0*가 선행하는 스레드 내의 다른 접근 사건들의 수행을 모두 중지시킨다. 그 이유는 *r0*가 선행하는 다른 모든 접근 사건들은 *r0*에 의해서 모두 영향받은 사건들이기 때문이다. *r8*이 발생하면 마찬가지로 쓰기 후보역사 내의 변두리 후보사건들과의 경합 발생 여부를 검사한다. 이 경우는 경합이 발생하였으므로 *r8*과 읽기 후보역사 내의 변두리 후보사건들과의 좌우 관계를 고려한 뒤에 새로운 변두리 후보사건이면 읽기 후보역사의 내용을 갱신하고, 이 사건을 읽기 후보사건으로 저장한다. 마지막으로 후보사건인 *r8*이 선행하는 스레드 내의 다른 접근 사건들은 *r8*에 의해서 모두 영향받은 사건들이기 때문에 현재의 스레드의 수행을 중지시킨다.

그림 1의 내포병렬성을 가진 프로그램을 수행한 결과인 표 2에서 쓰기 후보사건이 보고되지 않고 읽기 후보사건만이 보고되었다. 이러한 경우는 이 읽기 사건들이 얽힌 경합에 포함된 최초경합임을 나타낸다. 1단계의 CS(*X, Backup*)에 나타나고도, 2단계에서 쓰기 후보사건으로 보고되지 않는 경우는 그 쓰기 사건이 얽힌 경합에 포함되는 경우로서 읽기-쓰기 후보사건이다. 1단계의 CS(*X, Backup*)에는 얽힌 경합에 포함된 모든 쓰기 사건이 보고되는데, 최악의 경우에 기껏해야 하나의 읽기-쓰기 후보사건이 제외된다. 그 예로서 표 1을 보면, CS(*backup*)에 *w7*을 제외한 *w10*과 *w11*이 저장되어 있다.

2단계 탐지 프로토콜의 의미를 정리해 보면 다음과 같다. 첫째, (checkread_2nd()와 checkwrite_2nd()의 5번줄과 13번줄) 경합에 포함된 임의의 읽기나 쓰기 사건이 읽기나 쓰기 후보사건일 필요충분조건은 이 사건에 선행하면서 경합에 포함된 접근 사건이 존재하지 않고 이 사건과 경합인 변두리 후보사건이 존재하는 것이다. 둘째, (checkread_2nd()와 checkwrite_2nd()의 1-3번줄과 7-11번줄) 임의의 읽기나 쓰기 후보사건이 변두리 후보사건일 필요충분조건은 이 사건이 변두리 사건이고 이 사건에 선행하는 다른 읽기나 쓰기 변두리 사건이 존재하지 않는 것이다. 셋째, (checkread_2nd()의 13번줄) 임의의 읽기 후보사건이 존재하고 이 사건이 선행하는 다른 사건이 존재하지 않으면, 이 사건이 선행하는 읽기-쓰기 후보사건은 존재하지 않는다. 넷째, (checkwrite_ 2nd()의 13번줄) 임의의 쓰기 후보사건이 존재하고 이 사건이 선행하는 다른 사건이 존재하지 않으면, 이 사건이 선행하는 어떤 후보사건도 존재하지 않는다.

[정리 3.4] 경합에 포함된 임의의 접근사건이 읽기 (혹은 쓰기) 후보사건일 필요충분조건은 이 사건에 선행하면서 경합에 포함된 사건이 존재하지 않고 이 사건과 경합인 변두리 후보사건이 존재하는 것이다.

[정리 3.5] 임의의 읽기 (혹은 쓰기) 후보사건이 변두리 후보사건일 필요충분조건은 이 사건이 변두리 사건이고 이 사건에 선행하는 같은 접근 유형의 다른 변두리 사건이 존재하지 않는 것이다.

[정리 3.6] 임의의 읽기 혹은 쓰기 후보사건이 존재하고 이 사건이 선행하는 다른 사건이 존재하지 않으면, 읽기-쓰기 후보사건을 포함하여 이 사건이 선행하는 어떤 후보사건도 존재하지 않는다.

4. 프로토콜 분석

본 절에서는 앞 절에서 소개한 2단계 최초경합 탐지 프로토콜의 정당성을 입증하기 위해서 제시된 정리들을 증명하고, 그 효율성을 보이기 위해서 프로토콜의 수행 시에 요구되는 시간 및 공간 복잡도의 측면에서 분석한다.

4.1 정당성 증명

[정리 3.1] 임의의 수행에서 경합이 존재하면, 적어도

하나의 쓰기 사건이나 외부에 존재하는 읽기 사건 e_j가 경합에 포함되고, e_j와 경합에 포함되는 접근 사건 e_k와 경합에 포함되면서 e_j에 선행하는 접근 사건 e_i의 집합이 존재한다.

증명: 이 정리의 증명은 Mellor-Crummey 프로토콜[10]을 제안한 문헌에서 수행되었다. □

[정리 3.2] 경합에 포함된 임의의 접근 사건이 존재할 때, 이 사건이 변두리 사건이고 이 사건에 선행하는 다른 동일한 접근 유형의 변두리 사건이 존재하지 않으면, 이 사건은 변두리 후보사건이다.

증명: 이 정리의 대우정리인 '경합에 포함된 임의의 접근 사건 e_j가 존재할 때, e_j가 변두리 후보사건이 아니면, e_j는 변두리 사건이 아니거나 e_j에 선행하는 다른 동일한 접근 유형의 변두리 사건 e_i가 존재한다.'를 증명한다. 경합에 포함된 임의의 사건이 변두리 후보사건이 아닌 경우는 (A) e_j가 변두리에 있지 않은 후보사건이거나 (B) e_j가 후보사건이 아닌 경우이다. (A)의 경우에는 변두리 사건의 정의에 의하여 e_j가 변두리에 있지 않은 후보사건이면 내부에 존재하는 후보사건이므로 e_j는 변두리 사건이 아니다. (B)의 경우에는 경합에 포함된 임의의 접근 사건 e_j가 후보사건이 아니면, 정의에 의해서 경합에 포함된 e_j에 선행하는 임의의 접근사건 e_i가 존재하고, e_i는 e_j와 동일한 접근 유형인 경우도 포함된다. 그러므로 명제는 만족된다. □

[정리 3.3] 경합에 포함된 임의의 쓰기 사건이 존재할 때, 이 사건에 선행하며 경합에 포함된 어떤 쓰기 사건도 존재하지 않으면, 이 쓰기 사건은 후보사건이다.

증명: 후보사건의 정의에 의해서 간단히 증명된다. □

[정리 3.4] 경합에 포함된 임의의 접근사건이 읽기 (혹은 쓰기) 후보사건일 필요충분조건은 이 사건에 선행하면서 경합에 포함된 사건이 존재하지 않고 이 사건과 경합인 변두리 후보사건이 존재하는 것이다.

증명: (->) '경합에 포함된 임의의 접근사건이 읽기 (혹은 쓰기) 후보사건이면, 이 사건에 선행하면서 경합에 포함된 사건이 존재하지 않고 이 사건과 경합인 변두리 후보사건이 존재한다.'를 증명한다. 이 명제를 증명하기 위하여 대우명제인 '경합에 포함된 임의의 접근사건 e_j에 선행하면서 경합에 포함된 사건 e_i가 존재하거나 e_j와 경합인 변두리 후보사건이 존재하지 않으면, e_j는 읽기 (혹은 쓰기) 후보사건이 아니다.'를 증명한다. e_j에 선행하면서 경합에 포함된 사건 e_i가 존재한다면 e_j는 e_i에 영향받은 사건이므로 정의에 의해서 읽기 (혹은 쓰기) 후보사건이 아니다. e_j와 경합인 변두리 후보사건이 존재하지 않는 경우는 (A) e_j와 경합인 e_k가 변두리 후보사건이 아닌 경우이거나 (B) 변두리 후보사건이 e_j와 경합이 아닌 경우 등의 두 경우 외에는 없다. (A)의 경우에는 다른 변두리 후보사건 e_y가 존재하여 e_j와 경합을 이룰 수 있으므로 'e_j와 경합인 변두리 후보사건이 존재하지 않으면'이라는 전제에 모순이 된다. 그러므로 이 경우는 존재하지 않는다. (B)의 경우에는 (1) 변두리 후보사건이 e_j에 선행하거나 (2) e_j와 변두리 후보사건이 병행하지만 서로 대립되지 않은 경우이다. (1)의 경우에는 변두리 후보사건의 정의에 의해서 변두리 후보사건이면 경합에 포함된 사건이다. 변두리 후보사건이 e_j에 선행하면, e_j가 영향받은 사건이 되므로 e_j는 읽기 (혹은 쓰기) 후보사건이 아니다. (2)의 경우에는 e_j와 변두리 후보사건이 병행하지만 서로 대립되지 않으므로, 변두리 후보사건이 쓰기 사건인 경우가 존재하지 않는다. 그러면 변두리 후보사건과 e_j가 모두 읽기 사건이 되어 e_j와 경합에 포함되는 사건이 존재하지 않는다. 그러므로 이 경우는 '경합에 포함된 임의의 접근사건 e_j'라는 전제에 모순이 된다. 그러므로 이 경우는 존재하지 않는다.

(<-) '임의의 접근사건에 선행하면서 경합에 포함된 사건이 존재하지 않고 이 사건과 경합인 변두리 후보사건이 존재하면, 이 사건은 경합에 포함된 읽기 (혹은 쓰기) 후보사건이다.'를 증명한다. 이 명제를 증명하기 위하여 대우명제인 '경합에 포함된 임의의 접근사건 e_j가 읽기 (혹은 쓰기) 후보사건이 아니면, e_j에 선행하면서 경합에 포함된 사건 e_i가 존재하거나 e_j와 경합인 변두리 후보사건이 존재하지 않는다.'를 증명한다. 경합에 포함된 임의의 접근사건 e_j가 읽기 (혹은 쓰기) 후보사건이 아니면, 정의에 의해서 e_j에 선행하는 e_i가 존재하고 e_j와 경합을 이루는 접근 사건 e_k가 존재한다. 내포병렬성에서 e_j가 e_k와 경합에 포함되고 e_j에 선행하는 e_i가 존재하면 (A) e_i도 e_k와 경합인 경우이거나 (B) e_i가 e_k에 선행하는 경우이다. (A) 경우에는 e_j에 선행하면서 경합에 포함된 사건 e_i가 존재하므로, 명제가 만족된다. (B) 경우에는 정의에 의해서 e_k도 읽기 (혹은 쓰기) 후보사건이 아니므로 e_j와 경합인 변두리 후보사건이 존재하지 않는다. 그러므로 명제는 만족된다. □

[정리 3.5] 임의의 읽기 (혹은 쓰기) 후보사건이 변두리 후보사건일 필요충분조건은 이 사건이 변두리 사건이고 이 사건에 선행하는 같은 접근 유형의 다른 변두리 사건이 존재하지 않는 것이다.

증명: (->) '임의의 읽기 (혹은 쓰기) 후보사건 e_j가 변두리 후보사건이면, e_j는 변두리 사건이고 e_j에 선행하는 같은 접근 유형의 다른 변두리 사건이 존재하지 않는다.'를 증명한다. 이를 증명하기 위하여 대우명제인

'임의의 읽기 (혹은 쓰기) 후보사건 e_j가 변두리 사건이 아니거나 e_j에 선행하는 같은 접근 유형의 다른 변두리 사건이 존재하면, e_j는 변두리 후보사건이 아니다.'를 증명한다. 후보사건 e_j가 변두리 사건이 아니라면 e_j는 내부에 존재하는 사건이므로 변두리 후보사건의 정의에 의해서 e_j는 변두리 후보사건이 아니다. 후보사건 e_j에 선행하는 같은 접근 유형의 다른 변두리 사건이 존재하면, 후보사건의 정의에 의하여 e_j는 변두리 후보사건이 아니다.

(<=) '임의의 읽기 (혹은 쓰기) 후보사건 e_j가 변두리 사건이고 e_j에 선행하는 같은 접근 유형의 다른 변두리 사건이 존재하지 않으면, e_j는 변두리 후보사건이다.'를 증명한다. 이 명제는 [정리 3.1]에 의하여 만족된다. □

[정리 3.6] 임의의 읽기 혹은 쓰기 후보사건이 존재하고 이 사건이 선행하는 다른 사건이 존재하지 않으면, 읽기-쓰기 후보사건을 포함하여 이 사건이 선행하는 어떤 후보사건도 존재하지 않는다.

증명: 후보사건의 정의에 의해서 간단히 증명된다. □

4.2 효율성 분석

본 논문에서 제안된 기법은 각 접근 사건의 수행 시에 요구되는 비효율성을 개선하고자 한 것이다. 이러한 효율성은 요구되는 기억 공간과 접근역사 내의 사건들과의 비교 횟수에 의존적이며, 이러한 요소는 다시 각 접근역사를 구성하는 항목의 수와 각 항목에 저장되는 병행성 정보의 크기에 의존적이다. 본 기법에서는 적용되는 병행성 정보의 생성 및 적용을 위하여 2.3절에서 소개된 바와 같이 내포병렬성을 갖는 병렬프로그램에서 스레드간 병행성 정보를 수행 중에 가장 효율적으로 생성하는 NR 레이블링 기법을 이용한다. 이 기법을 적용하면, 접근역사의 각 항목에 저장되는 접근 사건 정보인 병행성 정보의 크기가 상수적이므로 각 접근역사를 위한 기억 공간은 접근역사를 구성하는 항목 수에 의존적이 된다.

프로그램의 최대병렬성을 T, 프로그램의 내포깊이를 N이라 할 때, NR 레이블링 기법은 각 스레드의 레이블을 생성하는데 소요되는 시간복잡도는 $O(N)$이고, 모든 레이블을 저장하는데 $O(NT)$의 공간복잡도가 요구된다. 그리고 임의의 두 레이블을 비교하는데 소요되는 시간은 $O(\log_2 N)$의 시간복잡도가 요구되고, 비교되는 레이블 중에서 한 레이블은 $O(1)$ 공간복잡도의 스레드 식별자 정보만이 이용된다.

감시하는 공유변수의 수를 V라 할 때, 내포병렬성을 가진 병렬프로그램의 최초경합을 수행 중에 가장 효율적으로 탐지하는 기존의 기법[5]은 각 공유변수에 대한 접근역사의 항목 수가 $O(T)$이므로, 모든 공유변수들의 접근역사를 위해서는 $O(VT)$의 공간복잡도가 요구된다. 그러므로 모든 레이블을 저장하는데 필요한 공간을 포함한 전체 공간복잡도는 $O(VT+NT)$가 되고, $(V >> N)$이므로 이 복잡도는 $O(VT)$가 된다. 또한 매 접근사건을 감시할 때, 해당 접근역사 내의 모든 항목과 현재의 접근 사건을 비교하는데 $O(T\log_2 N)$의 시간복잡도가 요구된다.

그러나 본 기법은 각 공유변수에 대한 접근역사의 항목 수가 $O(1)$이므로, 모든 공유변수들의 접근역사를 위해서는 $O(V)$의 공간복잡도만이 요구된다. 그러므로 모든 레이블을 저장하는데 필요한 공간을 포함한 전체 공간복잡도는 $O(V+NT)$가 되어, 기존 기법의 공간복잡도인 $O(VT)$ 보다 효율적이다. 또한 매 접근사건을 감시할 때, 해당 접근역사내의 모든 항목과 현재의 접근 사건을 비교하는데 $O(\log_2 N)$의 시간복잡도만이 요구되므로, 기존 기법의 시간복잡도인 $O(T\log_2 N)$보다 효율적이다.

표 3 접근 사건 수행 시의 효율성 비교

탐지 기법	기억 공간	사건 비교 횟수
기존 기법 [5]	$O(T)$	$O(T)$
제안된 기법	$O(1)$	$O(1)$

5. 결 론

내포병렬성을 가진 병렬프로그램에서의 수행 중 최초경합을 탐지하는 기존의 기법은 두 번의 수행을 통하여 후보사건들의 집합을 수집하는데, 각 공유 변수마다 프로그램의 최대병렬성에 의존하는 크기의 접근역사를 유지해야 하므로 비효율적인 수행 시간과 기억 공간을 요구하였다. 이러한 비효율성을 개선하기 위하여, 본 논문에서는 기존의 기법과 달리 각 공유변수에 대한 접근 사건들의 역사를 상수적 크기로 유지하여, 각 접근 사건의 수행 시에 상수적 복잡도의 사건 비교 횟수와 기억 공간만을 요구하는 새로운 수행중 탐지기법을 제안하였다.

본 기법은 두 단계의 수행을 통하여 수행중 최초경합을 탐지하는데, 첫 번째 수행에서는 각 공유 변수의 모든 접근 사건들의 선행 관계와 좌우 관계를 비교하여 외부에 존재하는 사건으로 접근역사를 구성한다. 이 때 유지되는 접근 사건들의 역사는 상수 크기이다. 매 접근 사건들이 접근역사 내의 사건들과 경합을 이루면 이를 분석하고 변두리 후보사건이면 상수적 크기의 후보역사에 수집한다. 두 번째 수행에서는 각 접근 사건들을 감

시하면서 첫 번째 수행에서 수집된 변두리 후보사건들과의 선행 관계와 좌우 관계를 검사하여 최초경합을 구성하는 후보사건들의 집합을 완성하여 보고한다.

이와 같이 각 공유 변수에 대한 접근역사의 크기를 상수개로 제한함으로써 본 기법은 내포병렬성을 가진 공유메모리 병렬프로그램에서 각 접근 사건의 수행 시에 프로그램의 최대병렬성에 무관하게 상수적 복잡도의 사건 비교횟수와 기억공간만으로 수행중 최초경합 탐지를 보장한다. 그러므로 본 기법은 내포병렬성을 가진 공유메모리 병렬프로그램의 디버깅을 보다 효율적이고 실용적이게 한다. 앞으로 남은 과제는 본 기법을 동기화가 있는 내포병렬 프로그램 모델로 확장하는 것이다.

참 고 문 헌

[1] Netzer R. H. B., and B. P. Miller, "What are Race Conditions? Some Issues and Formalizations," *Letters on Programming Lang. and Systems*, 1(1): 74 88, ACM, 1992.

[2] Choi J., and S. Min, "Race Frontier Reproducing Data Races in Parallel Program Debugging," *3rd Symp. On Principles and Practice of Parallel Programming*, pp. 145 154, ACM, 1991.

[3] Jun Y., and C. E. McDowell, "On the fly Detection of the First Races in Programs with Nested Parallelism," *2nd Int'l Conf. on Parallel and Distributed Processing Techniques and Applications*, pp. 1549 1560, *CSREA*, 1996.

[4] Kim J., and Y. Jun, "Scalable On the fly Detection of the First Races in Parallel Programs," *12nd Int'l. Conf. on Super computing*, pp. 345 352, ACM, 1998.

[5] Park H., and Y. Jun, "Detecting the First Races in Parallel Programs with Ordered Synchronization," *6th Int'l Conf. on Parallel and Distributed Systems*, pp. 201 208, IEEE, 1998.

[6] Ronsse M., K. De Bosschere, "RecPlay: A Fully Integrated Practical Record/Replay System," *Tr. on Computer Systems*, 17(2): 133 152, ACM, May 1999.

[7] Kim J., D. Kim, and Y. Jun, "Scalable Visualization for Debugging Races in OpenMP Programs," *3rd Int'l Conf. on Communications in Computing*, pp. 259 265, Las Vegas, Nevada, World Academy of Science, June 2002.

[8] Jun Y., and K. Koh, "On the fly Detection of Access Anomalies Nested Parallel Loops," *3rd Workshop on Parallel and Distributed Debugging*, pp. 107 117, ACM, 1993.

[9] Leslie Lamport, "Time, Clocks, and the Ordering of Events in Distributed System," *Comm. of the ACM*, 21(7): 558-565, ACM, July 1978.

[10] Mellor Crummey J., "On the fly Detection of Data Races for Programs with Nested Fork-Join Parallelism," *Supercomputing*, pp. 24 33, ACM/IEEE, Nov. 1991.

[11] Netzer R. H. B., and Miller B. P., "Improving the Accuracy of Data Race Detection," *3rd Symp. on Principles and Practice of Parallel Programming*, pp. 133 144, ACM, 1991.

[12] Chandra R., L. Dagum, D. Kohr, D. Maydan, J. McDonald, R. Menon, *Parallel Programming in OpenMP*, Academic Press, 2001.

[13] OpenMP Architecture Review Board, *OpenMP Fortran Application Program Interface*, Ver. 2.0, Nov. 2000.

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

[15] Park S., M. Park, and Y. Jun, "A Comparison of Scalable Labeling Schemes for Detecting Races in OpenMP Programs," *Int'l Workshop on OpenMP Applications and Tools, Lecture Notes in Comptuter Science*, 2104: 68 80, Springer Verlag, OpenMP ARB, July 2001.

하 금 숙

1983년 경북대학교 전자공학과(컴퓨터전공) 학사. 1990년 경북대학교 컴퓨터공학과 석사. 2003년 8월 경북대학교 컴퓨터공학과 박사. 1983년~1985년 한국컴퓨터(주) 연구원. 1986년~1992년 경북대학교 전자공학과 조교. 1992년~현재 구미1대학 컴퓨터전공 교수. 관심분야는 병렬 및 분산처리, 운영체제, 데이타베이스

전 용 기

1980년 경북대학교 컴퓨터공학과 학사
1982년 서울대학교 컴퓨터공학과 석사
1993년 서울대학교 컴퓨터공학과 박사
1982년~1985년 한국전자통신연구원(ETRI) 연구원. 1995년~1996년 미국 캘리포니아 주립대학교 (UCSC) 연구원
1985년~현재 경상대학교 컴퓨터과학과 교수. 관심분야는 병렬 및 분산 처리, 운영체제, 프로그래밍 시스템

유 기 영

정보과학회논문지 : 시스템 및 이론
제 30 권 제 1 호 참조