기술 보고서 번호 2005-04

실용적이고 효율적인 잠금없는 쓰레기 수거 참조 카운팅 기준

안데르스 기덴 스탐 **마리나 파파 란타 필루** **하칸 선델**
필립스 티가

컴퓨터 공학과
찰머스 공과 대학 및 예테보리 대학교
SE-412 96 예테보리, 스웨덴

예테보리, 2005

컴퓨터 공학 및 기술 보고서
찰머스 공과 대학 및 예테보리 대학교

기술 보고서 번호 2005-04
ISSN : 1652-926X

컴퓨터 공학과
찰머스 공과 대학 및 예테보리 대학교
SE-412 96 예테보리, 스웨덴

2005 년 3 월 스웨덴 예테보리

3 페이지

추상

우리는 효율적이고 실용적인 잠금없는 구현을 제시합니다.
참조를 기반으로 가비지 수집 체계의tation
임의 잠금없는 동적 사용을위한 계수
데이터 구조. 이 제도는 로의 안전을 보장합니다
전역 참조뿐만 아니라cal, 임의 메모리 지원
재사용, 모드에서 사용 가능한 원자 프리미티브 사용
컴퓨터 시스템에 상한을 제공하고
재사용이 불가능한 메모리. 우리가 아는 한에,
이것은이 모든 것을 제공하는 최초의 잠금없는 알고리즘입니다.
속성. 실험 결과는 상당한 성능을 나타냅니다
잠금없는 데이터 구조를 개선하여
강력한 쓰레기 수거를 요구하십시오.

키워드 : 참조 횟수, 가비지 수집, 잠금
무료, 공유 메모리.

1. 소개

동적 관리를 위해서는 메모리 관리가 필수적입니다
동시 데이터 구조. 데이터에 대한 동시 알고리즘
구조와 관련 메모리 관리는 일반적으로
상호 배제를 기반으로합니다. 그러나 상호 배제
차단의 원인이되며 결과적으로 심각한 문제가 발생할 수 있습니다.
교착 상태, 우선 순위 반전 또는 기아와 같은 렘. 레-
검색 자들은 이러한 문제를 해결함으로써
비 차단 동기화 알고리즘 (비 차단 동기화 알고리즘)
상호 배제를 기반으로합니다. 잠금이없는 알고리즘은
차단하고 항상 하나 이상의 작업을 보장합니다
회의에서 취한 조치와 관계없이 진행할 수 있습니다.
현재 작업. Wait-free [3] 알고리즘은 잠금이 없으며
또한 모든 작업이
행동에 관계없이 자신의 단계의 유한 수
동시 작업에 의해 수행됩니다. 비 중요하다
동시 작동의 영향을 미치는 차단 알고리즘
관련 프로세스에 의해
현명한 태도. 일반적인 일관성 요구 사항은
선형 화성이라고한다.

이 백서에서는 실용적이고 효율적으로 중점을두고 있습니다.
잠금없는 동적 환경에서 메모리 관리
데이터 구조. 알고리즘의 작동을 잠 그려면
모든 하위 작업은 잠금이 없어야합니다. 결과
결과적으로 잠금없는 동적 데이터 구조에는 일반적으로
잠금없는 메모리 관리. 메모리 관리
문제는 일반적으로 장애의 하위 문제로 나뉩니다.
namic 메모리 할당 대 가비지 수집. 발 루아
마이클과 스콧 (22, 13)뿐만 아니라
고정 크기 메모리 세그먼트에 대한 광 할당 방식;
이 구성표는
응답 가비지 수집 체계. 잠금없는 메모리
일반적인 사용을위한 할당 계획은
Michael [12] 및 Gidenstam et al. [2].

다양한 잠금이없는 가비지 수집 체계가
문헌에 발표. 마이클 [10, 11]은
로컬 참조에 중점을 둔 위험 포인터 알고리즘. 에이
유사한 체계가 Herlihy et al. [5]; 이
체계는 무제한 태그를 사용하며 이중
너비 CAS 원자 프리미티브, 비교 및 스왑 오페라
인접한 두 개의 메모리 워드를 원자 적으로 업데이트 할 수있는
일부 32 비트 아키텍처에서만 사용 가능하지만
현재 64 비트 아키텍처 중 거의 없습니다. 이것들로
체계는 지역 포인터의 안전을 보장합니다.
스레드, 그들은 임의의 잠금없는 알고리즘을 지원할 수 없습니다
항상 전역을 신뢰할 수 있어야하는 연산
참조 (즉, 데이터 구조 내에서 포인터)
사물. 이 제약은 강력하고 제한적일 수 있으며
데이터 구조 알고리즘이 순회를 재 시도하도록합니다.
큰 데이터 구조로 인해 큰 성능
동시성 수준에 따라 벌칙이 부과됩니다.

참조를 기반으로하는 가비지 콜렉션 체계
계산은 지역뿐만 아니라 세계의 안전을 보장 할 수 있습니다
객체에 대한 참조. Valois et al. [22, 13]은 자물쇠를 제시했다.
우리가 구현할 수있는 무료 참조 계산 체계
사용 가능한 원자 기본 요소를 사용하지만
메모리에 대한 해당 알고리즘에만 사용
위치. Detlefs et al. [1]는 다음을 허용하는 체계를 제시했다
재생 메모리의 임의 재사용
더블 워드 비교 및 스왑 연산 인 DCAS
두 개의 임의 메모리를 원자 적으로 업데이트 할 수있는
현대 건축에서는 사용할 수없는 단어입니다.
Herlihy et al. [4, 15]는 이전의 수정을 제시했다.
CAS (비교 및 스왑) 만 사용하도록 구성
참조 카운팅 파트. 그러나이 체계는
그 자체에는 이중 폭 CAS [1]이 필요한 다른 체계에서 .
참조 계수 기술은 [13]에서 확인되었다.
niques는 잠재적으로 스레드에서
블록 (재귀 참조 작성 기능으로 인해)
회수 할 노드의 수

대기없는 메모리 관리와 관련하여 대기
Valois의 체계의 자유로운 연장은 Sun에 의해 제시되었습니다
델 [18, 17]. Hesselink and Groote [7, 8]은
다음으로 제한되는 대기없는 메모리 관리 체계
토큰 공유의 특정 문제.

이 논문은 참조 계수의 강도를 결합
위험 포인터의 효율성으로 일반 잠금 장치
유지를 목표로 무료 참조 계산 체계
관련된 기술의 장점 만 피하면서
각각의 결점. 새로운 자물쇠가없는 쓰레기
수집 체계는 잠금이없고 선형화 가능하며,
메모리 할당을위한 임의의 체계로 가능
일반적으로 사용 가능한 원자 기본 요소를 사용하여 구현
그리고 글로벌 참조뿐만 아니라 지역의 안전을 보장합니다

[1] 두 개의 인접 항목을 원자 적으로 업데이트 할 수있는 비교 및 스왑 작업
센트 메모리 단어는 일부 32 비트 아키텍처에서 사용할 수 있지만
현재 64 비트 아키텍처 중 거의 없습니다.

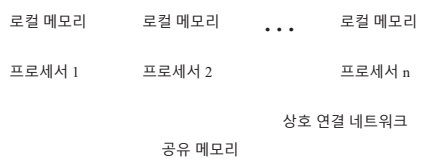

그림 1. 공유 메모리 멀티 프로세서 시스템 온도 구조

ences. 또한 메모리 양을 제한하는 방법을 보여줍니다
스레드에 의해 일시적으로 유지 될 수 있습니다.
나머지 논문은 다음과 같이 구성되어 있습니다. 섹션에서
2 우리는 구현 한 시스템의 유형을 설명합니다
목표로하고 있습니다. 섹션 3은 문제의 세부 사항을 설명합니다
우리가 집중하고있는 쓰레기 수거의 틀. 실제
알고리즘은 섹션 4에 설명되어 있습니다. 섹션 5에서는 다음을 정의합니다.
구현에 대한 작업의 정확한 의미
을 증명하고 알고리즘의 정확성을 보여줍니다.
잠금이없고 선형화 가능한 속성뿐만 아니라
일시적 일 수있는 메모리의 상한
우리의 제도에 의해 회수하기 위해 개최되었습니다. 섹션 6은
새로운 계획의 실험 평가
잠금이없는 데이터 구조. 우리는 Sec-
7.

2 시스템 설명

공유 메모리 멀티의 전형적인 추상화
프로세서 시스템 구성은 그림 1에 나와 있습니다.
시스템의 각 노드에는 프로세서와
로컬 메모리. 모든 노드는 공유에 연결됩니다
상호 연결 네트워크를 통한 메모리. 공동 세트
운영 작업이 시스템에서 실행 중입니다.
각각의 작업. 각 작업은 순차적으로 실행됩니다
각 프로세서는
한 번에 많은 작업을 수행합니다. 협력 작업, 가능성
다른 프로세서에서 실행, 공유 데이터 객체 사용
공유 메모리에 내장되어 조정 및 통신
케이트. 작업은 공유 데이터에 대한 작업을 동기화합니다
캐시 코 히어 런트 위에서 하위 작업을 통한 객체
공유 메모리. 그러나 공유 메모리는 그렇지 않을 수 있습니다
시스템의 모든 노드에 균일하게 액세스 할 수 있습니다. 프로세스
sors의 다른 부분에서 서로 다른 액세스 시간을 가질 수 있습니다
기억.
공유 메모리 시스템은 원자 읽기를 지원해야합니다
단일 메모리 워드뿐만 아니라 쓰기 작업
동기화를위한 더 강력한 원자 기본 요소. 이 pa에서
Fetch-and-Add (FAA) 및 Compare-
And-Swap (CAS) 원자 기본 요소; 자세한 내용은 그림 2를 참조하십시오.
scription. 이러한 읽기-수정-쓰기 스타일의 작업은

```
절차 FAA (주소 : 단어에 대한 포인터 , 수 : 정수 )
        원자력
            * 주소 : = * 주소 + 번호;

함수 CAS (주소 : word에 대한 포인터 , oldvalue : word ,
  새 값 : 단어 ) : 부울
        원자력
            경우 * = 어드레스는 OLDVALUE 후
                * 주소 : = 새로운 가치;
                true를 반환 ;
            그렇지 않으면 false를 반환합니다 .
```

그림 2. Fetch-and-Add (FAA) 및 CAS (Compare-and-Swap) 원자 기본 요소.

가장 일반적인 아키텍처에서 사용 가능하거나 쉽게
다른 동기화 프리미티브로부터 도출 된 것 [14] [9].

3 문제 설명

이 논문에서 우리는 쓰레기 충돌을 해결하는 것을 목표로하고 있습니다
동적 잠금없는 데이터 구조와 관련된 문제
tures. 잠금없는 데이터 구조는 일반적으로 세트로 구성됩니다.
각각 arbi-를 포함하는 *노드* 라고하는 메모리 세그먼트
트레이 리 데이터. 이 노드들은 참조하여 상호 연결됩니다
임의의 패턴으로 서로. 참고 문헌은 일반적으로
각각을 식별 할 수있는 *포인터* 를 사용하여 구현
메모리 주소에 의한 개별 노드. 마다
node는 임의의 수의 포인터를 포함 할 수 있습니다.
링크 는 다른 노드를 참조합니다. 따라야 할 작업
링크를 통해 참조 된 노드를 *역 참조* 라고 합니다.
일부 노드는 일반적으로 항상 데이터 구조의 일부입니다.
다른 모든 노드는 데이터 구조의 일부입니다.
자체가 데이터 구조의 일부인 노드에서 참조
사실. 동적이고 동시적인 데이터 구조에서 임의의
노드를 지속적으로 동시에 추가하거나 다시 추가 할 수 있습니다.
데이터 구조에서 이동했습니다. 시스템이 제한됨에 따라
메모리 양,이 노드의 점유 된 메모리
/에서 동적으로 할당 및 회수해야 함
시스템.
데이터 구조의 순차적 구현에서
노드의 메모리는 일반적으로
마지막 참조가 제거되었을 때의 시스템, 즉
노드가 *삭제* 되었을 때 . 동시 환경에서
이것은 또한 가능한 지역 참조를 포함해야합니다
스레드가 가질 수있는 노드
교정 된 노드의 메모리는 해당 노드에 치명적일 수 있습니다.
데이터 구조 및 / 또는 전체 시스템의 정확성. 그만큼
교정을 올바르게 결정하는 논리 단위를 호출합니다.
가비지 컬렉터 따라서 다음 있어야
재산:

속성 1 *가비지 수집기는 pos-*
데이터 구조의 일부가 아닌 Sible 쓰레기

	보장 공유의 안전 참조 (속성 5)	경계 수 교정되지 않은 삭제 노드 (속성 2)	호환 표준 메모리 할당 자 (속성 4)	충분 한 단어 비교 및 스왑
새로운 알고리즘	예	예	예	예
Detlefs et al. [1]	예	아니 전자	예	없음 해당 없음
Herlihy et al. [5]	아니	예	예	아니 b
Herlihy et al. [4, 15]	예	아니 전자	예	아니 c
마이클 [10, 11]	아니	예	예	예 d
Valois et al. [22, 13]	예	아니 전자	아니	예

a LFRC 알고리즘은 DCAS (Double-word Compare-and-Swap) 원자 기본 요소를 사용합니다.
b 패스-벅 (PTB) 알고리즘은 이중 폭 비교 및 스왑 원자 기본 요소를 사용합니다.
c SLFRC 알고리즘은 PBT (pass-the-buck) 알고리즘을 기반으로하므로 2 배 폭 비교 및 스왑을 사용합니다.
d 위험 포인터 알고리즘은 원자 읽기 및 쓰기 만 사용합니다.
e 이 참조 횟수 기반 체계를 사용하면 서로 반복적으로 참조하는 삭제 된 노드의 임의의 긴 체인을 만들 수 있습니다. 게다가,
주기적으로 서로 참조하는 삭제 된 노드 (예 : 순환 가비지)는 다시 회수되지 않습니다.

표 1. 비 차단 메모리 관리에 대한 다양한 접근 방식의 속성

스레드에 의한 향후 액세스가 불가능합니다.

항상 최대 값을 예측할 수 있어야합니다
데이터 구조에 사용되는 메모리 양
가비지 수집기에이 요구 사항을 추가합니다.

속성 2 *언제든지 상한이 있어야합니다*
데이터 구조의 일부가 아닌 노드 수
그러나 아직 시스템을 되찾지 못했습니다.

가비지 수집기 (GC)의 실제 구현
이러한 특성은 지역 심판으로 달성하기가 매우 어려울 수 있습니다.
전 세계적으로 노드에 액세스 할 수없는 경우 (예 :
프로세서 레지스터에 저장 될 수 있습니다). 따라서 간단합니다
GC의 언급은 일반적으로
볼드 스레드 및
노드, 예. 역 참조를위한 특수 작업을 제공함으로써
링크를하고 데이터 구조를 구현하도록 요구
노드가있을 때 tation은 명시 적으로 가비지 콜렉터를 호출합니다.
삭제되었습니다.
또한, 관심있는 기본 데이터 구조로
잠금이 없으며 일반적으로 선형화 가능합니다.
수집기는 다음 기능도 보장해야합니다.

속성 3 *가비지 수집기의 모든 작업*
기본 데이터 구조와의 간단한 통신
멘토는 잠금이없고 선형화되어야합니다.

전체 시스템의 총량을 최소화하기 위해
다양한 데이터 구조를위한 메모리를 차지했습니다.
시간은 다음과 같은 속성을 충족시키고 싶습니다 :

속성 4 *가비지가 회수하는 메모리*
수집기는 임의의 미래에 접근 할 수 있어야합니다
재사용; 즉 가비지 수집기는
시스템의 기본 메모리 할당 자

동시 환경에서 자주 발생할 수 있습니다
스레드가있는 노드에 대한 로컬 참조를 보유하고 있음

구조 노드
mm_ref : **정수** / * 초기에 0 * /
mm_trace : **부울** / * 처음에는 false * /
mm_del : **부울** / * 처음에는 false * /
... / * 임의의 사용자 데이터 및 링크는 * /를 따릅니다.
link [NR_LINKS_NODE] : 노드에 대한 **포인터** / * 초기 NULL * /

그림 3. 노드 구조

일부에 의해 삭제 (즉, 데이터 구조에서 제거)
다른 실. 이러한 경우에는
삭제 된 노드의 링크를 사용할 수있는 첫 번째 스레드 (예 :
큰 데이터 구조의 검색 절차 :

속성 5 *노드에 대한 로컬 참조가있는 스레드*
또한 모든 링크를 역 참조 할 수 있어야합니다.
해당 노드에 포함되어 있습니다.

이 백서의 새로운 알고리즘은 이러한 모든 제안을 충족시킵니다.
원 자성 Primi- 만 사용하는 특성 외에 다른 특성
현대 시스템에서 일반적으로 사용 가능한 tives. 표
1은 수행 된 특성과 이전 특성의 비교를 보여줍니다.
잠금이없는 가비지 수집 체계를 제공했습니다. 모든
구성표 중 속성 1과 3을 충족하는 반면
다른 속성의 하위 집합은 이전의
보낸 계획.

4 새로운 잠금없는 알고리즘

섹션에서 요청 된 모든 속성을 충족 시키려면
3 효율적이고 실용적인 방법을 제공합니다.
목표는 다음과 같은 참조 계산 방법을 고안하는 것입니다.
Michael 의 *위험 포인터* (HP) 체계 도 사용합니다.
[10, 11]. 대략적으로 말하면, 위험 포인터는
지역 참조 및 참조의 안전 보장
내부 링크의 안전을 보장하기 위해

6 페이지

데이터 구조. 따라서 각 노드의 참조 카운트
전 세계적으로 액세스 가능한 링크 수를 표시해야합니다.
해당 노드를 참조하십시오. 그림 3은 노드 구조를 설명합니다
알고리즘에서 사용됩니다. HP 체계에서와 같이
스레드는 삭제되었지만 삭제되지 않은 노드 목록을 유지합니다.
아직 회수되었으며이 목록은 재판매 가능성이 있는지 스캔됩니다.
길이가 특정 임계 값 (예 :
THRESHOLD_2). 삭제 된 노드 중 일부는
고정 된 수로 인해 교정이 방지 됨
위험 포인터, 일부 삭제 된 노드는 사전에
양의 기준 카운트로 인해 배출됨
모래밭. 따라서 참조 수를 유지하는 것이 중요합니다
따라서 링크에서 삭제 된 노드를 최소한으로 만듭니다. 우리 전에
의 크기를 제한하는 기술을 계속
삭제 목록, 우리는 무엇에 대한 가정을 소개합니다
잠금없는 데이터 구조 알고리즘에 필요합니다.

가정 1 *삭제 된 노드의 각 링크에 대해*
삭제 된 노드를 참조하면 교체 할 수 있어야합니다.
의미론이 유지되는 활성 노드에 대한 참조
관련된 스레드 중 하나에 대해.

이 가정의 직관은 관찰의 배후에 있습니다.
삭제 된 노드의 링크가 유용한 이유
로컬 참조가있는 스레드에 의한 참조 해제 그만큼
삭제 된 노드에 대한 로컬 참조가있는 스레드는 반드시 원합니다.
적절한 활성 노드를 찾아서
링크의 유리한. 해당 참조가 또한
여기서 삭제 된 노드에 이전 단계가 반복됩니다. 에서
관심있는 스레드의 관점에서 볼 수는 없습니다.
다른 스레드가 절차에 도움이 된 경우 차이점
이미 삭제 된 노드의 링크가 있는지 확인했습니다.
모든 참조 활성 노드. 교체 절차
활성 노드에 대한 참조가있는 삭제 된 노드의 링크는
*정리*라고 *합니다*.

앞에서 설명한 것처럼 위험 포인터 외에도 노드의
삭제 목록이 교정되지 않을 수 있음
다른 삭제 된 노드의 링크로. 이 노드는
동일한 삭제 목록 또는 다른 스레드의 삭제 목록에 있습니다.
이러한 이유로 모든 스레드의 삭제 목록에 액세스 할 수 있습니다.
스레드에 의해 읽습니다. 삭제 목록의 길이가
스레드가 특정 임계 값 (THRESHOLD_1)에 도달
삭제 목록에있는 모든 노드를 정리합니다. 모든 경우
노드 중 여전히 교정이 방지됩니다.
다른 스레드의 삭제 목록에있는 노드로 인해
따라서 스레드는 다른 모든 것을 정리하려고 시도합니다.
스레드의 삭제 목록도 있습니다. 이 절차가 반복됨에 따라
삭제 목록의 길이가 임계 값 이하가 될 때까지
오래된, 아직 교정되지 않은 삭제 된 노드의 양
경계가 있습니다. THRESHOLD_1의 실제 계산
부록 5.2에 설명되어 있습니다. 임계 값 THRESH-
OLD_2가 HP 체계 이하로 설정되었습니다.
THRESHOLD_1보다.

4.1 응용 프로그래밍 인터페이스

다음 기능은 안전한 취급을 위해 정의됩니다.
참조 카운트 노드 :

DeRefLink **함수** (링크 : 노드에 **대한 포인터**) :
노드에 대한 **포인터**
절차 ReleaseRef (노드 : 노드에 대한 **포인터**)
기능 CompareAndSwapRef (링크 : **포인터 포인터**
노드, 이전 : 노드에 대한 **포인터** , 노드 : 노드에 대한 **포인터**) : **부울**
절차 StoreRef (링크 : **포인터 포인터** 노드,
노드 : 노드에 대한 **포인터**)
NewNode **기능** : Node에 대한 **포인터**
DeleteNode (node : Node에 대한 **포인터) 프로 시저**

DeRefLink 함수 는 주어진 *것을* 안전하게 역 참조합니다
링크 해제하고 참조 해제 된 노드에 대한 위험 포인터를 설정합니다.
따라서 반환 안전에 대한 미래의 안전 보장
마디. 프로 시저 *ReleaseRef*를 호출해야 할 때
주어진 노드는 현재 스레드에 의해 액세스되지 않습니다.
더. 해당 위험 포인터가 지워집니다.

동시에있을 수있는 링크를 업데이트하려면
링크에 대한 업데이트, *CompareAndSwapRef* 함수
업데이트 사용 여부에 대한 결과를 제공하는
성공 여부. 절차는
링크에서 *DeRefLink* 를 호출하는 스레드 는 안전하게 그렇게 할 수 있습니다.
스레드에 노드에 대한 위험 포인터 참조가있는 경우
링크가 포함되어 있습니다. 요구 사항은
CompareAndSwapRef의 스레드 에는 위험이 있어야합니다.
주어진 주어진 노드에 대한 포인터.

동의가없는 링크를 업데이트하려면
임대료 업데이트 *StoreRef* 를 호출해야합니다. 그만큼
프로시 저는 *DeRe-* 를 호출하는 모든 스레드가
스레드에 위험이있는 경우 링크의 *fLink* 가 안전하게 그렇게 할 수 있습니다.
링크가 포함 된 노드에 대한 ard 포인터 참조.
요구 사항은 *StoreRef* 의 호출 스레드입니다 .
주어진 노드에 대한 위험 포인터가 있어야합니다.
저장되고 다른 스레드는 아마도
현재 링크에 있습니다 (그렇지 않으면 *CompareAndSwapRef*
대신 호출해야합니다).

NewNode 함수 는 새로운 노드를 할당하고
미래의 안전을 보장하는 무료 위험 포인터
액세스 한 다음 반환합니다. *DeleteN-* 절차
노드가 데이터에서 제거 될 때 *ode* 를 호출해야합니다.
구조와 회수 할 수있는 메모리
재사용을 위해. *DeleteNode* 라는 사용자 작업 은
삭제 된 노드에 대한 모든 참조를 제거하는 책임
데이터 구조의 활성 노드에서 이것은 유사하다
메모리 할당자를 사용할 때 필요한 것보다 lar
순차적 데이터 구조. 메모리 관리자는
안전하게 삭제 될 때까지 삭제 된 노드를 재 확보하십시오.

4.4 절에서 이러한 기능이 어떻게 작동하는지에 대한 예를 제공합니다.
잠금없는 큐 알고리즘의 컨텍스트에서 사용될 수 있습니다.
연결된 목록을 기반으로합니다.

7 페이지

```
/ * 전역 변수 * /
HP [NR_THREADS] [NR_INDICES] : 노드에 대한 포인터 ;
DL_Nodes [NR_THREADS] [THRESHOLD_1] : 노드에 대한 포인터 ;
DL_Claims [NR_THREADS] [THRESHOLD_1] : 정수 ;
DL_Done [NR_THREADS] [THRESHOLD_1] : 부울 ;
/ * 위의 행렬은 다음의 값으로 초기화되어야합니다.
NULL, NULL, 0 각각 false * /

/ * 지역 정적 변수 * /
threadId : 정수 ; / * 각 스레드마다 고유하고 고정 된 번호
  0과 NR_THREADS-1 * / 사이
dlist : 정수 ; / * 처음에는 ⊥ * /
dcount : 정수 ; / * 처음에는 0 * /
DL_Nexts [THRESHOLD_1] : 정수 ;

/ * 지역 임시 변수 * /
node, node1, node2, old : Node에 대한 포인터 ;
스레드, 인덱스, new_dlist, new_dcount : 정수 ;
plist : Node에 대한 포인터 배열 .

DeRefLink 함수 (링크 : 노드에 대한 포인터 ) :
  노드에 대한 포인터
D1    HP [threadId] [index] = NULL이되도록 색인을 선택하십시오.
D2    진실한 동안
D3        노드 : = * 링크;
D4        HP [threadId] [index] : = 노드;
D5        만약 * = 링크 노드 후
D6            리턴 노드;

절차 ReleaseRef (노드 : 노드에 대한 포인터 )
R1    HP [threadId] [index] = node가되도록 색인을 선택하십시오.
R2    HP [스레드 ID] [인덱스] : = NULL;

CompareAndSwapRef (link : Node에 대한 포인터를 가리키는 함수 ,
  구 : 포인터 : 노드, 노드 에 대한 포인터 노드) : 부울
C1    만약 CAS (링크, 오래 된, 노드) 다음
C2        만약 노드 = NULL 다음
C3            FAA (& node.mm_ref, 1);
C4            node.mm_trace : = 거짓 ;
C5        만약 이전 =가 null 다음 FAA (old.mm_ref, -1);
C6        true를 반환 ;
C7    거짓 반환 ;

절차 StoreRef (링크 : 포인터 포인터 노드,
  노드 : 노드에 대한 포인터 )
S1    이전 : = * 링크;
S2    * 링크 : = 노드;
S3    만약 노드 = NULL 다음
S4        FAA (& node.mm_ref, 1);
S5        node.mm_trace : = 거짓 ;
S6    만약 이전 =가 null 다음 FAA (old.mm_ref, -1);

NewNode 기능 : Node에 대한 포인터
NN1 node : = 노드 의 메모리 할당 ( 예 : malloc 사용)
NN2 노드 .mm_ref : = 0;
NN3 node.mm_del : = 거짓 ;
NN4 HP [threadId] [index] = NULL이되도록 인덱스를 선택하십시오.
NN5 HP [threadId] [index] : = 노드;
NN6 리턴 노드;
```

그림 4. 참조 카운팅 기능, 파트 I

```
DeleteNode (node : Node에 대한 포인터) 프로 시저
DN1 ReleaseRef (노드);
DN2 node.mm_del : = true ; node.mm_trace : = 거짓 ;
DN3 DL_Nodes [threadId] [index] = NULL과 같은 색인을 선택하십시오.
DN4 DL_Done [스레드 ID] [인덱스] : = 거짓;
DN5 DL_Nodes [threadId] [index] : = 노드;
DN6 DL_Nexts [색인] : = dlist;
DN7 dlist : = 색인; dcount : = dcount + 1;
사실이지만 DN8
DN9       만약 DCOUNT = THRESHOLD_1 다음 CleanUpLocal ();
DN10      만약 DCOUNT ≥ THRESHOLD_2 다음 검사 ();
DN11      만약 DCOUNT = THRESHOLD_1 다음 CleanUpAll ();
DN12      그렇지 않으면 휴식 ;
```

그림 5. 참조 카운팅 기능, 파트 II

```
절차 TerminateNode (node : Node에 대한 포인터 , 동시 : boolean )
TN1 하지 않을 경우 동시 접속 후
TN2       모든 X 여기서 링크 [X]의 노드는 레퍼런스 카운트 DO
TN3           StoreRef (node.link [x], NULL);
TN4 그 외
TN5       모든 X 여기서 링크 [X]의 노드는 레퍼런스 카운트 DO
TN6           반복 node1 : = node.link [x];
TN7           까지 CompareAndSwapRef (node.link [X], 노드 1, NULL);

절차 다음 cleanupNode (노드 : 포인터 노드)
CN1 모든 X 링크 [X]의 노드는 레퍼런스 카운트 DO
        다시 해 보다:
CN2       node1 : = DeRefLink (& node.link [x]);
CN3       만약 노드 1 = NULL 과 node1.mm_del 다음
CN4           node2 : = DeRefLink (& node1.link [x]);
CN5           CompareAndSwapRef (& node.link [x], node1, node2);
CN6           ReleaseRef (노드 2);
CN7           ReleaseRef (노드 1);
CN8           고토 재시도;
CN9       ReleaseRef (노드 1);
```

그림 6. 콜백 함수

콜백

다음 함수는 콜백이어야합니다.
각 특정 데이터 구조의 설계자에 의해 벌금이 부과됩니다.

절차 다음 cleanupNode (노드 : **포인터** 노드)
절차 TerminateNode (node : Node에 대한 **포인터** , 동의 함)
임대 : **부울**)

TerminateNode 프로시 저는 다음 을 확인합니다.
지정된 노드의 링크 중 어느 것도
다른 노드. 삭제 된 노드에서 *TerminateNode* 가 호출 됨
다른 노드 또는 스레드로부터 클레임이없는 경우
노드. 2

CleanUpNode 프로시 저는 모든
주어진 노드의 링크에서 주장 된 참조는

2 원칙적으로이 절차는 메모리 관리자가 제공 할 수 있습니다.
그러나 실제로는 사용자가 메모리를 결정하게하는 것이 더 편리합니다
노드 레코드의 레이아웃. 모든 노드 레코드는 여전히 시작해야합니다
mm_ref, mm_trace 및 mm_del 필드와 함께.

8 페이지

```
CleanUpLocal () 프로 시저
CL1 지수 : = dlist;
CL2 동안 인덱스 = ⊥ 할 일
CL3         노드 : = DL_Nodes [threadId] [index];
CL4         CleanUpNode (노드);
CL5         index : = DL_Nexts [인덱스];

CleanUpAll () 프로 시저
CA1 에 대한 스레드 : = 0 에 NR_THREADS-1 DO
CA2         에 대한 인덱스 : = 0 에 THRESHOLD_1-1 할 일
CA3         node : = DL_Nodes [스레드] [인덱스];
CA4         만약 노드 = NULL 이 아닌 DL_Done [스레드] 인덱스] 다음
CA5             FAA (& DL_Claims [스레드] [인덱스], 1);
CA6             만약 노드 DL_Nodes [스레드] [지수 =] 다음
CA7                 CleanUpNode (노드);
CA8             FAA (& DL_Claims [스레드] [인덱스],-1);

Scan () 절차
SC1 지수 : = dlist;
인덱스 = ⊥ 수행 하는 동안 SC2
SC3         노드 : = DL_Nodes [threadId] [index];
SC4         만약 node.mm_ref = 0 다음
SC5             node.mm_trace : = true ;
SC6             만약 node.mm_ref = 0 다음 node.mm_trace = 거짓 ;
SC7         index : = DL_Nexts [인덱스];
SC8 plist : = ∅; new_dlist : = ⊥ ; new_dcount : = 0;
SC9 에 대한 = 0 ; 스레드 에 NR_THREADS-1 DO
SC10        에 대한 인덱스 : = 0 에 NR_INDICES-1 DO
SC11            node : = HP [스레드] [인덱스];
SC12            만약 노드 = NULL 다음
SC13                plist : = plist + 노드;
SC14 배열 plist 에서 중복 정렬 및 제거
dlist = ⊥ do 동안 SC15
SC16        인덱스 : = dlist;
SC17        노드 : = DL_Nodes [threadId] [index];
SC18        dlist : = DL_Nexts [인덱스];
SC19        만약 node.mm_ref = 0 및 node.mm_trace 및 노드 ∈ PLIST 후
SC20            DL_Nodes [threadId] [index] : = NULL;
SC21            만약 DL_Claims는 [threadId는 [색인] = 0 다음
SC22                종단 노드 (노드, 거짓);
SC23                노드의 메모리를 비 웁니다
SC24                계속 ;
SC25            종단 노드 (노드, 참);
SC26            DL_Done [threadId] [index] : = true;
SC27            DL_Nodes [threadId] [index] : = 노드;
SC28        DL_Nexts [색인] : = 새로운 _ 목록;
SC29        new_dlist : = 인덱스;
SC30        new_dcount : = new_dcount + 1;
SC31 dlist : = new_dlist;
SC32 dcount : = new_dcount;
```

그림 7. 내부 기능.

활성 노드 만 가리 키므로 중복 pas-
임의의 수의 삭제 된 노드를 통해 처리합니다.

4.2 보조 절차

내부 사용을 위해 정의 된 보조 기능
참조 계산 체계 :

Scan () **절차**
CleanUpLocal () **프로 시저**
CleanUpAll () **프로 시저**

스캔 이 아직 다시 검색되지 않은 모든 절차 를 검색합니다.
이 스레드에 의해 삭제 된 청구 된 노드와 해당 노드 만 회수
일치하는 위험 포인터가없고
노드 내부의 링크에서 계산 된 참조가 있습니다.
절차 *CleanUpLocal는* redun-을 제거하려고합니다
dant는 삭제 된 노드의 링크에서 참조를 주장했습니다.
이 스레드에 의해 삭제되었습니다. 절차 *CleanU-*
pAll 은 중복 주장 된 참조를 제거하려고 시도합니다.
스레드에 의해 삭제 된 삭제 된 노드의 링크

4.3 자세한 알고리즘 설명

DeRefLink (그림 4)는 먼저 노드에 대한 포인터를 읽습니다.
라인 D3의 * link에 저장됩니다. 그런 다음 D4 행에서 다음 중 하나를 설정합니다.
스레드의 위험 포인터가 노드를 가리 킵니다. 라인 D5에서
링크가 여전히 이전과 동일한 노드를 가리키는 지 확인합니다.
* link가 여전히 노드를 가리키는 경우 노드가
여전히 아직 교정되지 않았으며이를 재생할 수 없습니다
위험 포인터가 이제 그것을 가리키는 때까지. 만약
* 링크는 마지막으로 읽은 이후 변경되어 다시 시도합니다.

ReleaseRef (그림 4).이 스레드의 위험을 제거합니다
노드를 가리키는 포인터. 노드 노드가
삭제, 참조 횟수가 0이고 다른 위험이 없습니다.
포인터가 가리키면 노드를 다시 사용할 수 있습니다.
스캔에 의해 (*DeleteNode* 를 호출 한 스레드에 의해 호출 됨)
노드).

CompareAndSwapRef (그림 4)는 일반적인
링크의 CAS는
그에 따라 각 노드. C4 라인은 모든 동의를 통지합니다
임대 *스캔* 노드의 기준 카운트 IN-되었음을
주름진. 노드 노드는 다음 중 액세스가 안전합니다.
ING *CompareAndSwapRef는* 이후 스레드가 호출 *COM-을*
pareAndSwapRef 에는 위험 포인터가 있어야합니다.
그것을 가리키는. 라인 C5에서 오래된 기준 카운트는
감소했습니다. 이전 참조 횟수는
* link가 이전 노드를 참조 했으므로 0보다 큽니다.

StoreRef (그림 4)는 다음과 같은 경우에만 사용할 수 있습니다.
* link의 동시 업데이트가 없습니다. 에서 * link를 업데이트 한 후
라인 S2 *StoreRef* 는
호출 스레드 때문에 안전 라인 S4, *StoreRef가* 있다
노드 노드에 대한 위험 포인터가 있어야합니다. 선
S5는 동시 *스캔* 에게 참조 카운트가
노드가 0이 아닙니다. 라인 S6에서 오래된 기준 카운트
감소합니다. 이전 참조 횟수는
* link가 이전 노드를 참조 했으므로 0보다 큽니다.

NewNode (그림 4), 새 노드에 메모리를 할당합니다
기본 메모리 할당 자에서
각 노드의 헤더 필드 또한 위험을 설정합니다
노드에 대한 포인터.

DeleteNode (그림 5), 노드 노드를 logi-로 표시
DN2 행에서 cally가 삭제되었습니다. 그런 다음 DN3에서 DN7까지
이 스레드의 삭제 세트에 노드가 삽입되었지만 삭제되지 않았습니다.

9 페이지

아직 재개 된 노드. DN4 행에서 DL_Done을 지워서
DN5 동시 *CleanUpAll* 라인에 포인터 쓰기
오퍼레이션은 노드에 액세스하여 참조를 정리할 수 있습니다.
이 스레드 세트에서 삭제 된 노드 수가
삭제되었지만 아직 회수되지 않은 노드가 크거나 같습니다.
THRESHOLD_2 회수 하는 *스캔* 이 수행됩니다.
다른 노드에서 참조하지 않는 세트의 모든 노드 또는
스레드.
스레드 세트가 삭제되었지만 재 확보되지 않은 노드 세트 인 경우
이제 가득 찼습니다. 즉 THRESHOLD_1 노드가 포함되어 있습니다.
스레드는 먼저 DN9 행에서 *CleanUpLocal* 을 실행 하여
삭제 된 모든 노드가 노드를 가리키는 지 확인하십시오.
CleanUpLocal 이 시작될 때 *활성화* 되었습니다. 그런 다음 *스캔* 을 실행합니다.
DN10 라인에서. *스캔* 이 노드를 전혀 회수 할 수없는 경우
스레드가 *CleanUpAll* 을 실행 하여
모든 스레드의 삭제 된 노드 세트

콜백

TerminateNode (그림 6), 노드의 모든 링크를 지워야합니다
노드의 링크에 NULL을 작성하여 노드. 이건 끝났어
사용하여 하나 *CompareAndSwapRef* 또는 *StoreRef* 디
동시 업데이트가 있을지 여부에 따라 보류 중
이 링크의 유무.
CleanUpNode (그림 6)는
노드 노드의 링크가 삭제 된 노드를 가리킴
CleanUpNode 호출이 시작 되기 전에

보조 절차

스캔 (그림 7), 현재에 의해 삭제 된 모든 노드를 회수
다른 노드 또는 위험 요소가 참조하지 않는 스레드
ard 포인터. 삭제 된 노드를 확인하려면
안전하게 회수 할 수 있습니다. *스캔* 먼저 mm_trace 비트를 설정합니다
참조 카운트가 0 인 모든 삭제 된 노드 중
SC1에서 SC7까지). SC6 라인에서 점검하면
따라서 mm_trace 비트가 0 일 때 카운트는 실제로 0이었습니다
세트.
그런 다음 *스캔* 은 모든 활성 위험 포인터를 기록합니다
plist의 스레드 (SC8-SC14 행). 라인 SC15에서
SC30 *스캔* 은 아직 재개되지 않은 모든 노드를 탐색합니다.
이 스레드에 의해. 이러한 각 노드에 대해 라인 테스트
SC19는 i) 기준 카운트가 0인지를 결정하고, ii)
참조 카운트는
위험 포인터를 읽었습니다 (mm_trace 비트로 표시됨)
설정) 및 iii) 노드가 위험에 의해 참조되지 않음
바늘. 이 세 가지 조건이 모두 해당되면 노드는
참조되지 않고 *스캔* 이 동시에 있는지 확인
CleanUpAll 의 라인 SC21에서 노드에서 작업을 작업.
그러한 *CleanUpAll* 작업 이없는 경우 *Scan* 은 *Ter*
minateNode 는 노드가 *연결할* 수있는 모든 참조를 해제합니다.
노드를 오염시킨 다음 재생합니다 (SC22 및 SC23 라인). 에서
CleanUpAll 작업 이 동시에 수행 될 수있는 경우
중단 노드 *스캔* 은 동시 버전의 *Ter를* 사용합니다.
minateNode 는 모든 노드의 링크를 NULL로 설정합니다. 으로
노드가 존재하기 전에 라인 SC26에서 DL_Done 플래그 설정
SC27 라인에서 교정되지 않은 노드 세트에 다시 삽입
나중에 *CleanUpAll* 조작으로이 노드를 막을 수 없습니다.
후속 *스캔* 에서 회수됩니다 .
CleanUpLocal (그림 7)은 스레드의 목록을 탐색합니다.
삭제되었지만 재 확보되지 않은 노드는 *다음* 에서 *CleanUpNode* 를 호출 *합니다* .
각 링크가 해당 링크를 참조하지 않도록
CleanUpLocal을 수행 할 때 이미 삭제 된 노드
시작했다.
CleanUpAll (그림 7)은 DL_Nodes를 통과합니다.
모든 스레드의 광선을 확인하고
참조는 노드에는 이미 있던 노드에 대한 링크가 포함되어 있습니다.
CleanUpAll 이 시작 되면 삭제됩니다 . 라인 CA4의 테스트
CleanUpAll 이 불필요하게 *스캔을* 방해 하지 않도록 방지
참조가없는 노드의 경우. 라인 테스트
CA6은 *CleanUpAll* 이 다음 노드에 액세스 하지 못하게 합니다.
스캔 이 이미 회수되었습니다. 노드가 여전히 존재하는 경우
라인 CA6의 DL_Nodes [thread] [index]
이 노드에 액세스하는 현재 *스캔* 은 라인 SC20 이전에 있어야합니다.
노드를 교정하지 않고 SC27 라인 이후에 있어야합니다.

4.4 적용 예

메모리 관리를위한 새로운 알고리즘의 적용
동적 데이터 구조를위한 잠금없는 알고리즘에 대한 노화
이전과 비슷한 방식으로 똑바로 진행될 수 있습니다.
많은 메모리 관리 계획을 제시했습니다. 그림 8
Valois 등의 잠금없는 큐 알고리즘을 보여줍니다. [21, 13]
mem-에 대한 새로운 알고리즘과 통합되므로
광석 관리.

4.5 알고리즘 확장

간단하게하기 위해이 논문의 알고리즘은
고정 된 수의 스레드로 설명됩니다. 그러나, 그
동적 수에 대한 체계를 쉽게 확장 할 수 있습니다.
HP 체계에 대해 설명 된 것과 유사한 방식으로 스레드
[11]. 글로벌 위험 포인터 매트릭스 (*HP*)는
연결된 배열 목록으로 바뀌 었습니다. 삭제 목록도
글로벌 체인에 연결되고 델리의 규모로
tion 목록 변경 사항, 오래된 중복 삭제 목록을 안전하게 사용할 수 있습니다.
메모리에 추가 HP 구성표를 사용하여 회수
조치.

4.6 알고리즘의 정확성과 경계 청구 된 메모리

정리 1 *이 알고리즘은 잠금이없고 Lin-*
가비지 수집을위한 이어 가능한 구성표.

정리 2 *삭제되었지만 아직 회수되지 않은 수*
시스템의 노드는

$$N^2 \cdot (k + l\,max + \alpha + 1) ,$$

10 페이지

```
구조 QNode
        mm_ref : 정수
        mm_trace : 부울
        mm_del : 부울
        다음 : QNode에 대한 포인터
        값 : 포인터 값

/ * 전역 변수 * /
head, tail : QNode를 가리키는 포인터

절차 InitQueue ()
IQ1 노드 : = NewNode ();
IQ2 node.next : = NULL;
IQ3 헤드 : = NULL; 꼬리 : = NULL;
IQ3 StoreRef (& head, node);
IQ4 StoreRef (& 꼬리, 노드);

Dequeue () 함수 : Value에 대한 포인터
DQ1 은 사실이지만
DQ2         node1 : = DeRefLink (& head);
DQ3         다음 : = DeRefLink (& node2.next);
DQ4         next = NULL 인 경우 NULL을 반환 합니다.
DQ5         경우 CompareAndSwapRef (머리, 노드 1, 다음) 후 휴식 ;
DQ6         ReleaseRef (노드 1); ReleaseRef (다음);
DQ7 DeleteNode (노드 1);
DQ8 값 : = next.value; ReleaseRef (다음);
DQ9 반환 값;

프로 시저 Enqueue (값 : Value에 대한 포인터 )
EQ1 노드 : = NewNode ();
EQ2 node.next : = NULL; node.value : = 값;
EQ3 old : = DeRefLink (& 테일); 이전 : = 오래된;
EQ4 반복
EQ5         prev.next = NULL을 수행 하는 동안
EQ6             prev2 : = DeRefLink (& prev.next);
EQ7             만약 이전 =이 이전 다음 ReleaseRef (이전);
EQ8             prev : = prev2;
CompareAndSwapRef (& prev.next, NULL, node); 까지 EQ9
EQ10 CompareAndSwapRef (& 테일, 이전, 노드);
EQ11 이 이전 = prev 이면 ReleaseRef (prev);
EQ12 ReleaseRef (이전); ReleaseRef (노드);
```

그림 8. 사용하는 큐 알고리즘의 예
새로운 메모리 관리 체계.

여기서 N은 시스템의 스레드 수이고, k는
스레드 당 위험 포인터 수, 최대 l 이 최대
노드가 포함 할 수있는 링크 수와 α는 최대
일시적으로 A를 가리킬 수있는 라이브 노드의 링크 수
삭제 된 노드.

위의 정리에 해당하는 증거가 남아 있습니다
5 절에서 일련의 렘-
마스.

5 정확성 증명

이 섹션에서 우리는 우리의 정확성을 증명합니다.
산술. 먼저 알고리즘이 되찾지 않음을 증명합니다
여전히 액세스 할 수있는 메모리, 우리는 입증
삭제되었지만 회수되지 않은 금액에 대해
쓰레기가있을 수 있으며 마지막으로 알고리즘이
선형화 가능하고 잠금이없는 것 [6]. 정의 세트
증거를 구조화하고 단축하는 데 도움이 될 것입니다.
이 섹션에 기록되어 있습니다. 우리는 순차적을 정의하는 것으로 시작합니다
우리 작업의 의미론.

정의 1 *F_t 가 자유 풀의 상태를 나타냅니다.*
시간 t에서 노드. 우리는 $n_t F_t$ 가 n 일 때 참으로 해석 합니다.
스캔의 라인 SC23에 따라 해제되었습니다. (바람직하게는
잠금 해제) 메모리 할당기를 사용하여 여유 공간을 관리 할 수 있습니다.
풀.
$n \in HP_t(p)$ 가 스레드 p에 검증 된 위험 요소가 있음을 나타냅니다.
시간 t에서 노드 n을 가리 키도록 설정된 ard 포인터. 검증 된 위험
포인터는 성공했거나 반환 한 것입니다.
ful DeRefLink 작업. 위험 포인터의 배열
구현. 어레이 HP는 포인트-
실패한 DeRefLink 작업으로 일시적으로 설정 한 경우
그러나 이것들은 $HP_t(p)$ 세트의 일부로 간주되지 않습니다.
$n \in DL_t(p)$ 가 노드 n이 삭제되고
시간 t에서 스레드 p의 dlist에서 교정을 기다리는 중.
$Del_t(n)$ 이 노드 n이 logi-
시간 t에서 cally가 삭제되었습니다. 삭제 표시가 제거되지 않습니다
노드가 사용 가능한 풀로 돌아올 때까지.
링크 (n) 가 공유 링크 세트 (포인터)를 표시하게하십시오
노드 n에 존재합니다.
$l_x \mapsto\rightarrow_t n_x$ 는 공유 링크 l_x 가
시간 t에서 노드 n_x.
$Ref_t(n)$ 이
시간 t에서 노드 n을 가리 킵니다. 공유 링크는 글로벌
응용 프로그램 또는 포인터 변수에 표시되는 공유 변수
노드 내부에 상주 할 수 있습니다. 구체적으로
위험 포인터의 스레드 별 배열, HP 및
삭제 된 노드의 스레드 배열 DL_Nodes는 고려되지 않습니다.
메모리 내부에 있기 때문에 공유 링크로 잘못됨
조치.
선형화에 관심이있는 작업
DeRefLink (DRL), ReleaseRef (RR), NewNode (NN),
DeleteNode (DN) 및 CompareAndSwapRef (CASR).
메모리 관리의 안전과 정확성
다음과 같은 추가 내부 작업을 노후화하십시오
또한 관심 분야 : 종단 노드 (TN), 스캔 (SCAN),
CleanUpNode (CUN), CleanUpLocal (CUL), CleanU-
pAll (CUA).
순차를 정의하는 다음 표현식에서
연산의 의미론에서 구문은 $S_1 : O_1, S_2$,
여기서 S_1 은 작업 O_1 이전의 조건부 상태입니다.
S_2 는 작업이 완료된 후의 결과 상태입니다.
수행.

DeRefLink

$$\exists n_1 . l_1 \mapsto\rightarrow_{t1} n_1 : \mathbf{DRL}(\mathbf{l_1}) = n_1, n_1 \in HP_{t2}(p_{curr}) \quad (1)$$

11 페이지

$$l\,1 \mapsto\!\!\rightarrow t\,1\ \bot : \mathbf{DRL}\,(\mathbf{l\,1}) = \bot, \tag{2}$$

출시

$$n \in HP\,t\,1\,(p\,curr) : \mathbf{RR}\,(\mathbf{n\,1}),\ n / \in HP\,t\,2\,(p\,curr) \tag{삼}$$

뉴 노드

$$\begin{array}{l} \exists n\,1 . N\,1 \in F\,t\,1 : \\ \quad \mathbf{NN}\,() = \mathbf{n\,1}, \\ \qquad n\,1 / \in F\,t\,2 \wedge Ref\,t\,2\,(n\,1) = 0 \wedge \neg Del\,t\,2\,(n\,1) \\ \qquad \wedge n\,1 \in HP\,t\,2\,(p\,curr) \end{array} \tag{4}$$

DeleteNode

$$\begin{array}{l} n\,1 \in HP\,t\,1\,(p\,curr) : \\ \quad \mathbf{DN}\,(\mathbf{n은\ 1}), \\ \qquad Del\,t\,2\,(n\,1) \wedge n\,1 \in DL\,t\,2\,(p\,curr) \\ \qquad \wedge n\,1 / \in HP\,t\,2\,(p\,curr) \end{array} \tag{5}$$

CompareAndSwapRef

$$\begin{array}{l} l\,1 \mapsto\!\!\rightarrow t\,1\ \bot \wedge n\,2 \in HP\,t\,1\,(p\,curr) : \\ \quad \mathbf{CASR}\,(\mathbf{l\,1}, \bot, \mathbf{n\,2}) = \mathbf{True}, \\ \qquad l\,1 \mapsto\!\!\rightarrow t\,2\,n\,2 \wedge l\,1 \in Ref\,t\,2\,(n\,2) \wedge \\ \qquad n\,2 \in HP\,t\,2\,(p\,curr) \end{array} \tag{6}$$

$$\begin{array}{l} 1\,n\,1 . l\,1 \mapsto\!\!\rightarrow t\,1\,n\,1 \wedge n\,2 = n\,1 : \\ \quad \mathbf{CASR}\,(\mathbf{l\,1}, \mathbf{n\,2}, \bot) = \mathbf{True}, \\ \qquad l\,1 \mapsto\!\!\rightarrow t\,2\ \bot \wedge l\,1 / \in Ref\,t\,2\,(n\,2) \end{array} \tag{7}$$

$$\begin{array}{l} 1\,n\,1 . l\,1 \mapsto\!\!\rightarrow t\,1\,n\,1 \wedge n\,2 = n\,1 \wedge \\ n\,3 \in HP\,t\,1\,(p\,curr) \wedge l\,1 \in Ref\,t\,1\,(n\,2) : \\ \quad \mathbf{CASR}\,(\mathbf{l\,1}, \mathbf{n\,2}, \mathbf{n\,3}) = \mathbf{True}, \\ \qquad l\,1 \mapsto\!\!\rightarrow t\,2\,n\,3 \wedge l\,1 / \in Ref\,t\,2\,(n\,2) \wedge \\ \qquad l\,1 \in Ref\,t\,2\,(n\,3) \wedge n\,3 \in HP\,t\,2\,(p\,curr) \end{array} \tag{8}$$

$$\begin{array}{l} 1\,n\,1 . l\,1 \mapsto\!\!\rightarrow t\,1\,n\,1 \wedge n\,1 = n\,2 \wedge n\,3 \in HP\,t\,1\,(p\,curr) : \\ \quad \mathbf{CASR}\,(\mathbf{l\,1}, \mathbf{n\,2}, \mathbf{n\,3}) = \mathbf{False}, \\ \qquad l\,1 \mapsto\!\!\rightarrow t\,2\,n\,1 \wedge n\,3 \in HP\,t\,2\,(p\,curr) \end{array} \tag{9}$$

주사

:

$$\begin{array}{l} \mathbf{스캔}\,() , \\ \quad \forall n\,1 \in DL\,t\,1\,(P\,CURR) . (n\,i \in F\,t\,2 \wedge \\ \quad (\ \forall n\,x \text{ st } l\,x \mapsto\!\!\rightarrow t\,1\,n\,x \wedge l\,x \in 링크\,(n\,i) . \\ \quad L\,X / \in 참조\,t\,(2)\,(N\,(X))) \vee (\exists p\,J . N\,1 \in HP\,t\,1\,(P\,J)) \vee \\ \quad (\exists n\,J . N\,J / \in F\,t\,1 \wedge \exists X \in 링크\,(N의\,J) . L\,X \mapsto\!\!\rightarrow t\,1\,N\,1) \end{array} \tag{10}$$

TerminateNode (애플리케이션 프로그램에서 구현 됨 그래머).

$$\begin{array}{l} n\,1 \in DL\,t\,1\,(p\,curr) : \\ \quad \mathbf{TerminateNode}\,(\mathbf{n\,1}, \mathbf{c}) , \\ \qquad \forall x \in 링크\,(n\,1) . (l\,x \mapsto\!\!\rightarrow t\,2\ \bot \wedge \\ \qquad \forall n\,x \text{ st } l\,x \mapsto\!\!\rightarrow t\,1\,n\,x . l\,x \in 참조\,t\,2\,(n\,x)) \end{array} \tag{11}$$

CleanUpNode (응용 프로그램에서 구현 됨 그래머).

$$\begin{array}{l} \exists p\,I . N\,1 \in DL\,t\,1\,(P\,I) \wedge 델\,t\,1\,(n은\,1) \\ \quad \mathbf{CleanUpNode}\,(\mathbf{n\,1}) , \\ \qquad \forall x \in 링크\,(n\,1) . (l\,x \mapsto\!\!\rightarrow t\,2\ \bot \vee \\ \qquad (\exists n\,x . l\,x \mapsto\!\!\rightarrow t\,2\,n\,x \wedge \neg Del\,t\,1\,(n\,x)) \end{array} \tag{12}$$

CleanUpLocal

:

$$\begin{array}{l} \mathbf{CleanUpLocal}\,() , \\ \quad \forall n\,I \in DL\,t\,1\,(P\,CURR) . (\forall x \in 링크\,(n\,i)) . \\ \quad l\,x \mapsto\!\!\rightarrow t\,2\ \bot \vee (\exists n\,x . l\,x \mapsto\!\!\rightarrow t\,2\,n\,x \wedge \neg Del\,t\,1\,(n\,x)) \end{array} \tag{13}$$

CleanAll

:

$$\begin{array}{l} \mathbf{CleanUpAll}\,() , \\ \quad \forall p\,i . (\forall n\,J \in DL\,t\,1\,(P\,I) . (\forall X \in 링크\,(N\,J) . \\ \quad l\,x \mapsto\!\!\rightarrow t\,2\ \bot \vee (\exists n\,x . l\,x \mapsto\!\!\rightarrow t\,2\,n\,x \wedge \neg Del\,t\,1\,(n\,x))) \end{array} \tag{14}$$

StoreRef (노드에서 링크를 업데이트하는 데만 사용할 수 있음) 다른 모든 스레드에는 액세스 할 수 없습니다.)

$$\begin{array}{l} l\,1 \mapsto\!\!\rightarrow t\,1\ \bot \wedge n\,2 \in HP\,t\,1\,(p\,curr) : \\ \quad \mathbf{SR}\,(\mathbf{l\,1}, \mathbf{n\,2}) , \\ \qquad l\,1 \mapsto\!\!\rightarrow t\,2\,n\,2 \wedge l\,1 \in Ref\,t\,2\,(n\,2) \wedge n\,2 \in HP\,t\,2\,(p\,curr) \end{array} \tag{15}$$

$$\begin{array}{l} In\,1 . l\,1 \mapsto\!\!\rightarrow t\,1\,n\,1 \wedge n\,2 \in HP\,t\,1\,(p\,curr) : \\ \quad \mathbf{SR}\,(\mathbf{l\,1}, \mathbf{n\,2}) , \\ \qquad l\,1 \mapsto\!\!\rightarrow t\,2\,n\,2 \wedge l\,1 / \in Ref\,t\,2\,(n\,1) \wedge \\ \qquad l\,1 \in Ref\,t\,2\,(n\,2) \wedge n\,2 \in HP\,t\,2\,(p\,curr) \end{array} \tag{16}$$

정의 2 *시간 t 에서 노드 n 을* 회수 *할 수 있다고한다* $Ref\,t\,(n) = \emptyset$ *및* $\forall pn / \in HP\,t\,(p)$ *및* $Del\,t\,(n)$.

5.1 안전

Lemma 1 *시간 t 1 에서 노드 n 을* 회수 *할 수 있는 경우 모든* $t \geq t\,1$ *에 대해* $Ref\,t\,(n) = \emptyset$.

증명 : (스케치) 모순으로 가정합니다.
시간 $t\,1$ 에서 노드 n 을 재 확보 할 수 있으며 나중에 시간 $t\,2$ 에서 노드 n 을 재 확보 할 수 있습니다.
$참조\,(n)\,t\,2 = \emptyset$. 교정 가능한 다음의 정의에서
시간 $t\,1$ 에서 n*을* 가리키는 공유 링크가 없었 으며
어떤 프로세스도 n*에* 대한 위험 포인터를 갖지 않았습니다 .
그런 다음 n 은 공유 된 일부에 저장되어 있어야합니다.
시간 $t\,1$ 이후 및 $t\,2$ 이전의 링크 l . 두 가지만 있습니다
공유 링크를 업데이트 할 수있는 버전 : *StoreRef* (SR) 및 *CASR* (CompareAndSwapRef). 그러나이 두 가지
오퍼레이션은 오퍼레이션을 발행하는 프로세스가
당시 n 에 대한 위험 포인터 . 두 가지 경우가 있습니다.
i) 공정 이후 위험 포인터가 n*으로* 설정되었습니다.
시간 $t\,1$ 이전에 준비 . 이 없었기 때문에 이것은 불가능하다

12 페이지

시간 t_1 에서 n 에 대한 위험 포인터 .
ii) 프로세스가 나중에 위험 포인터를 n*으로* 설정
t_1 보다 . 위험을 설정하는 유일한 방법으로는 불가능합니다
기존 노드에 대한 포인터는 *DeRefLink* 작업입니다.
거부 할 n 을 가리키는 공유 링크가 있어야합니다.
그리고 시간 t_1 에서 그러한 링크는 없다. (또한 참조
*DeRefLink*에 대한 *선형성*증명)
따라서 사용할 수있는 프로세스가 없기 때문에
링크를 업데이트하기위한 *StoreRef* 또는 *CompareAndSwapRef*
가리킨 N 시간의 t_1 이상 우리는이 *참조*$_t$ (N) = $\emptyset$ 위한
모두 $t \geq t_1$. 2

보조 정리 2 *노드 n은 DeRefLink 의해 반환*(L_1) *OP-*
*프로세스 p에 의해 수행 된*포기 *는 재 확보 할 수 없습니다.*
*그리고 될 수 없다*재생 가능한 *해당 전에*
ReleaseRef (n) *작업은 동일한 프로*
운.

증명 : (스케치) *DeRe-를 수행*할 때 노드 n*을* 회수 할 수 *없습니다.*
p 가 위험 포인터를 가리 키도록 설정 했기 때문에 *fLink가* 반환 됩니다.
라인 D4에서 n . 또한 라인 D5는 n 이 참조 되는지 확인합니다.
하여 erenced L_1 유해성 포인터가 설정된 후에 또한 어느
n 사이에 n 을 회수 할 수 없도록 보장
n 은 여전히 공유 링크에 의해 참조 되므로 D3 및 D4 라인 . 34 2

Lemma 3 *mm_ref 필드와 위험 요소*
포인터는 Ref$_t$ (n) *세트 의 안전한 근사치를 제공합니다.*

증명 : (스케치) 각 기준 카운트 필드 mm_ref
노드는 n*을* 참조하는 링크 세트의 근사값입니다.
참조 (n). 노드의 예 mm_ref 필드와 N*이* 아니라
진행중인 5 명의 운전자 가 없을 때 정확성 보장
관련된 ations N . 변경 될 수있는 유일한 작업
노드 n 의 mm_ref 필드 는 *CompareAndSwapRef 입니다.*
및 *StoreRef* .
메모리 관리 체계의 중요한 측면
의 *참조* (N) 세트가 비어 또는 비인지를 아는 것입니다
노드가 교정 가능한지 여부를 판별하려면 비어 있습니다. 파
특히 중요한 경우는 mm_ref 필드가
기준 카운트 감소를 지연시키기 때문에 증가
메모리 관리의 안전을 손상시키지 않습니다
계획.
따라서, 저장 될 노드 n 의 mm_ref 필드는
CompareAndSwapRef 또는 *StoreRef*에 의한 공유 링크
같은 원자 시간 순간에 작업이 증가하지 않습니다

3 참고 1 : D3과 D5 사이에서 n 이 l_1 에서 멀어 졌을 수 있습니다.
그런 다음 다시 이동했습니다.
4 참고 2 : D3과 D4 사이에서 "원래" n 은 실제로
제거하고 회수 한 다음 동일한 메모리를 재사용 할 수 있습니다.
D5 이전 에 l_1에 저장된 새로운 노드 n . 이것은 문제가되지 않습니다
"새로운" n 은 DeRefLink가 실제로 반환하는 것입니다.
5 충돌 한 작업이 진행중인 것으로 간주하십시오.

작업이 적용 되므로 *문제가되지 않습니다*.
노드 이후 메모리 관리 체계의 안전성
작동하는 동안 명확하게 회수 할 수 없습니다
어쨌든 작업을 수행하는 프로세스는
위험 포인터를 n*으로* 설정해야 합니다. 2

Lemma 4 *Scan 작업은 절대 노드를 회수하지 않습니다.*
그것은 회수 할 수 없습니다.

증명 : (스케치) *스캔* 시 노드 n 을 회수한다고 합니다.
사용 가능한 메모리 풀로 반환됩니다.
라인 SC23.
한다고 가정 *스캔* 노드 회수 N 시간에서 t_3 및하자
시간 t_1 및 t_2 는 *스캔*실행 라인 SC5 의 시간을 나타내고
노드 n*에* 대한 라인 SC19 .
먼저, $n \in$ 과 같은 프로세스 p 가 존재하지 않음에 유의하십시오.
HP (*(P)* 사이의 전체 구간 동안) t_1 과 t_2 사람
이러한 위험 포인터는 *스캔에* 의해 감지됩니다 (라인 SC9
-SC13). 결과적으로 n 에 액세스 할 수있는 모든 프로세스
시간 후 t_3 은 (*DeRefLink 사용*) 역 참조되어야합니다 .
시간 t_1 이후 n 을 가리키는 공유 링크 .
둘째, *Scan* 이 노드를 회수하므로
라인 SC5에서 **true** 로 설정된 n 의 mm_trace 필드
줄에서 0으로 확인 된 mm_ref 필드
SC6은 여전히 라인 SC19에 도달했을 때 그 값을 가졌다.
이것은 다음을 의미합니다.
i) *StoreRef* 또는 *CompareAndSwapRef* 옵션 이 없습니다.
t_1 이전에 시작된 공유 링크에 n 을 저장하는 erations 1
위험이 다음을 가리 키기 때문에 t_2 전에 끝나지 않았습니다.
n 필요한 작업이 감지되었을 때
*스캔*은 라인 SC9-SC13에서 위험 포인터를 검색했습니다.
ii) *StoreRef* 또는 *CompareAndSwapRef* 가 없습니다
공유 링크에 n 을 저장 하는 작업
트윈 t_1 과 t_2 는 그러한 작업이 지워졌을 것입니다.
mm_trace 필드와 비교하여
SC19가 실패합니다.
따라서 진행중인 *StoreRef* 또는 *Compare-*
에서 공유 링크에 n 을 저장하는 *AndSwapRef* 작업
스캔이 SC5 행을 실행 한 시간
이러한 작업은
mm_ref 필드에는 *Ref*$_{t_1}$ (n) = $\emptyset$*가* 있습니다. 또한, 때문에
ii) *StoreRef* 또는 *Compare-*
AndSwapRef 작업은 n 을 공유 링크 에 저장 합니다.
t_1 과 t_2 사이에서 시작하고 끝났습니다 .
Ref$_{t_1}$ (n) = $\emptyset$ 이기 때문에 *DeRefLink* 작동을 찾을 수 없습니다
n 이 아닌 한 시간 t_1 이후에 n 을 성공적으로 역 참조함으로써 ish
시간 t_1 후에 공유 링크에 저장된다 . 그러나 우리가
이러한 저장 조작은 시간이 지나면 시작해야합니다.
t_1 및 시간 t_2 및 수행 프로세스 후 완료
단일 프로세스가 없다는 첫 번째 관찰에 의해
전체 동안 n*에* 대한 위험 포인터를 가질 수 있었다 .
사이 terval t_1 과 $t_{(2)}$, 역 참조있다 N 후 t_1 . 이
분명히 모순이며, 따라서 그것은 불가능하다
시간 t_1 이후에 n 을 성공적으로 역 참조하는 프로세스 .

위에서 우리는 $t \geq t_1$에 대해 $Ref_t(n) = \emptyset$ 및 $\forall pn / \in HP의_t(P용) t \geq t_2$ 및 그 이후 $t_3 > t_2 > t_1$ 이면, 시간 t_3 에서 n 을 회수 할 수있다 . 2

Lemma 5 *Scan 작업은 절대 노드를 회수하지 않습니다. 동시에 CleanUpAll 작업으로 액세스됩니다.*

증명 : (스케치)에 저장된 노드 n*을* 회수하기 전에 DL_Nodes 배열에서 위치 i 를 *스캔* 하면 NULL 이 기록됨 DL_Nodes [i] (라인 SC20)에 입력 한 후 DL_Claims [i]는 0이다 (라인 SC21).

노드를 액세스하기 전에 N *CleanUpAll* oper- ATION 읽 N DL_Nodes [에서 I IN- 후, (광고 CA3) DL_Claims [i] (CA5 줄)를 접은 다음 확인 DL_Nodes [i]는 여전히 n (라인 CA6)을 포함합니다.

이제 동시에 *CleanUpAll* 작업을 수행하려면 운이 N 는 모두 DL_Nodes [의 판독을 수행하는 I (광고 CA3 및 라인 CA5) *스캔이* 라인 SC20을 수행 하기 전에 DL_Claims [i]가 증가하고 (라인 CA3) *스캔* 이 SC21 라인에서이를 감지하고 노드를 회수하지 않습니다.

반면에 *Scan* 이 클레임 카운트 0을 읽는 경우 SC21 라인에 동시 *CleanUpAll* 작업이 수행됩니다. CA6 라인의 DL_Nodes [i]에서 NULL을 읽지 않습니다. 노드에 액세스하십시오. 2

5.2 삭제 된 수에 대한 한계 증명 그러나 교정되지 않은 노드

정리 3 *각 스레드 p 용 I 의 최대 개수 $DL(p_i)$ 에서 삭제되었지만 재 확보되지 않은 노드 는 최대 $N \cdot (k + l_{max} + \alpha + 1)$, 여기서 N은 스레드 수 시스템, k는 스레드 당 위험 포인터의 수입니다. l_{max} 는 노드가 포함 할 수있는 최대 링크 수입니다. α는 라이브 노드의 최대 링크 수입니다. 일시적으로 삭제 된 노드를 가리킬 수 있습니다. (숫자 de- 응용 프로그램에 따라 다릅니다.)*

증명 : (스케치) 크기를 늘리는 유일한 작업 의 $DL(P의 난)$ 인 *DeleteNode* 때 $DL|(P I)|$ 에 도달 THRESHOLD_1 시도하기 전에 *CleanUpAll* 을 실행 합니다. 노드를 교정하십시오.

먼저 동시성이없는 경우를 고려하십시오. 다른 스레드에 의한 *DeleteNode* 작업 그러고 나서 *CleanUpAll* 을 가리키는 삭제 된 노드가있을 수 없습니다 $DL(p_i)$의 노드에 왼쪽. 그래서, 무엇을 방지 할 수 *페이지 전을* 재에서 하나 개의 특정 노드를 주장 DL (*피의* I)? 노드가 i)이를 가리키는 위험 포인터 또는 ii) 일부 라이브 노드는 여전히이를 가리 킵니다. 링크 수 삭제 된 노드를 가리킬 수있는 활성 노드 α 메모리를 사용하여 응용 프로그램 데이터 구조 나이. 우리는 삭제하는 각 응용 프로그램 작업이 필요합니다 노드는 노드에서 해당 노드에 대한 모든 참조를 제거해야합니다. 데이터 구조의 실제 노드가 완료되기 전에 항상 최대 $N \cdot \alpha$ 링크 가 있는지 확인 삭제 된 노드를 가리키는 라이브 노드에서 따라서 부재시 동시 *DeleteNode* 조작의 최대 수 *스캔* 에서 교정 할 수없는 $DL(p_i)$ 의 노드 수는 $N \cdot (k + \alpha)$.

동시 *DeleteNode* 옵션 이있는 경우 다음과 같은 세 가지 중요한 사항이 발생할 수 있습니다. i) Addi- $DL(p_i)$ 의 노드에 대한 포인터를 보유 할 수있는 기본 노드 시작 후 삭제 될 수 있습니다 *CleanUpAll* 방지 임의의 노드를 회수하지 않고 *스캔* 하십시오. 그러나이 경우 p_i 는 *CleanUpAll* 을 다시 시도 하고 일부 문제가 *발생한* 후 다시 *스캔*하십시오. 현재 작업이 진행되었습니다. ii) 일부 동시 *DeleteNode* 작업이 지연되거나 충돌 할 수 있습니다. 라인 DN2와 DN5 간 또는 SC21과 SC22 간 삭제 된 노드 하나를 "숨겨"주는 *Scan* 호출 에서 $DL(p_i)$의 노드를 가리키는 링크를 포함 할 수 있습니다. P *난* 의 *CleanUpAll* . 이런 식으로 서로의 스레드가 환기 P*는 난을* 최대 회수에서 *리터 최대* 의 노드 $DL(P I를)$. iii) 마지막으로, 동시 *CleanUpAll* 작업으로 인해 P*는 내가* 그들을 주장하면서 재생 가능한 노드를 하자는에서 *CleanUpNode* 작업을 수행 합니다. 하나, 이러한 노드가 발생했을 경우 *페이지를 내가* 의 *스캔을* 사용합니다 *Termi-을 nateNode* 는 노드의 링크를 NULL로 설정하고 노드의 DL_Done 플래그로 향후 *CleanU- p 모든* 작업으로 노드의 *재전송* 을 방지 주장했다. 경우 p*는 내가* 재시도 필요 *스캔*, 그것은 단지 사전 될 수 있습니다 최대 N 개의 재생 가능 노드를 재생하지 못함 동안 동시에 *CleanUpAll* 으로 인해 회수하지 못했습니다. 이전 *스캔*.

따라서, 노드의 최대 번호 $DL(P 내가)$ 그 P*는 내가* 회수는 $NN(k + l_{max} + \alpha + 1)$ 보다 작을 수 없습니다 . 2

결과 1 *정리 임계 값 인 THRESHOLD_1 이 사용되었습니다. 알고리즘에 의해 $N \cdot (k + l_{max} + \alpha + 1)$ 로 설정되어야합니다 .*

Corollary 2 *삭제되었지만 아직 회수되지 않은 수 시스템의 노드는*

$$N^2 \cdot (k + l_{max} + \alpha + 1),$$

여기서 N은 시스템의 스레드 수이고, k는 스레드 당 위험 포인터 수, 최대 l 이 최대 노드가 포함 할 수있는 링크 수와 α는 최대 일시적으로 A를 가리킬 수있는 라이브 노드의 링크 수 삭제 된 노드.

5.3 선형성

Lemma 6 *DeRefLink (DRL* $(l_1) = n_1$ *) 작업은 다음과 같습니다. 원자.*

증명 : (스케치) *DeRefLink* (DRL) 작업을 직접 가지고 *tar* 와 *비교* 하는 *CompareAndSwapRef* (CASR) 와의 상호 작용 *Scan* 에서 동일한 링크와 메모리 교정을 얻습니다 . 에이

14 페이지

CASR 작업은 DRL 작업보다 먼저 적용됩니다.
라인 C1의 CAS 명령은 * link가 실행되기 전에 실행됩니다.
DRL의 D5 행을 읽습니다.
스캔 라인에 DRL하여 위험 포인터 세트를 읽고
설정된 후 D4는 다음에 의해 참조 해제 된 노드를 해제하지 않습니다.
DRL (*스캔의* SC19 라인의 테스트에서 이를 방지합니다.
현재 *스캔* 후 해당 위험 포인터를 읽습니다.
DRL에 의해 설정되면 노드를 해제하지 않습니다. *스캔이* 읽는 경우
DRL에 의해 설정되기 전 해당 위험 포인터
스캔 은 노드의 참조 카운트가 아닌 것을 감지합니다
스캔 실행 중 0 또는 0이 아닌 값
조작. *2*

Lemma 7 *ReleaseRef (RR* (*n* 1) *) 작업은 다음과 같습니다.*
원자.

증명 : (스케치) *ReleaseRef* 작업 이 다음에 적용됩니다.
노드에 대한 위험 포인터가 NULL로 설정된 경우 라인 R2. *2*

Lemma 8 *NewNode (NN* () = *n* 1 *) 작업은 다음과 같습니다.*
메모리 할당자가 풀을 관리하는 데 사용 된 경우
여유 메모리 자체는 선형화 가능합니다.

증명 : (스케치) *NewNode* 작업 은 다음과 같은 경우에 적용됩니다.
새 노드의 메모리가 풀에서 제거됩니다.
여유 메모리. 선형화 가능한 메모리 할당 기의 경우
잘 정의 된 시간 순간에 일어난다. *2*

보조 정리 9 *DeleteNode (DN* (*n은* 1) *연산이다)*
원자.

증명 : (스케치) *DeleteNode* 작업 이 적용됩니다.
DN2 행에서 노드가 삭제 된 것으로 표시 될 때 *2*

보조 정리 10 *CompareAndSwapRef*
(CASR (*l* 1 , *n* 1 , *n* 2) *) 연산은 원자 적입니다.*

증명: **(스케치)** 그만큼 *CompareAndSwapRef*
(*CASR* (*n* 1)) 작업은
다른 *CompareAndSwapRef* (CASR) 및 *DeRefLink*
동일한 링크와 메모리를 대상으로하는 작업
스캔 에서 회수 .
CASR 작업은 CAS 인스트럭션시 적용됩니다.
C1 라인에서 실행됩니다. *2*

렘마 11 *스캔에 의해 삭제 된 노드 n의 교정*
원자입니다.

증명 : (스케치) *스캔* 시 노드 *n* 을 회수한다고 합니다.
사용 가능한 메모리 풀로 반환됩니다.
라인 SC23. 테스트에서 노드를 회수하는 것이 안전합니다.
SC19 라인은 i) *스캔* 이 위험 포인터를 찾지 못했음을 보증합니다
노드를 가리키고, ii) 노드의 참조 카운트
위험 포인터 이전부터 지속적으로 0
스캔되었습니다. ii) 유지는 n.mm_trace에 의해 보장됩니다.
비트, *스캔* 시 노드 *n* 이 *Ref* (*n*) = 0 *이었는지* 감지
라인 SC4를 실행했지만 나중에 포인터가
위험 포인터가 없도록 공유 링크 변수
스캔이 SC9 ~ SC13 라인에서 읽을 때 그때,
이 호출 *스캔* 을 되찾기 위해 시도하지 않습니다
노드에 대한 공유 링크가 다시 제거 되더라도
n.mm_trace가 포인터를 지우면
노드가 공유 변수에 저장되었습니다 (*Compare-*
AndSwapRef 또는 *StoreRef*).
SC21 라인의 테스트는 다른 것이 없음을 보장합니다.
CleanU-의 일부로 노드에 액세스 할 수있는 스레드
모든 작업. *CleanUpAll의* 동작 검증한다 그
청구를 증가시킨 후 노드가 여전히 라인 CA6에 있습니다.
노드에 액세스하기 전에 CA5 라인에서 카운터하십시오.
스캔이 SC21 행에서 클레임 수 0을 읽는 경우
액세스 할 수있는 동시 *CleanUpAll* 작업이 없습니다 .
CA6 라인에서 NULL을 읽을 것이기 때문에 노드. *2*

정리 4 *알고리즘은 선형화를 구현합니다*
가비지 수집기.

증명 : 이것은 Lemmas 6, 7, 8, 9 및 10에서 따릅니다. *2*

5.4 잠금없는 재산의 증거

렘마 12 *동기에 의한 재시도*
한 번의 작업으로 항상 작업이 진행됩니다.
다른 동시 작업에 의한 동작

증명 : 이제 가능한 실행 경로를 검사합니다.
이행. *ReleaseRef* , *NewNode* 작업
그리고 *CompareAndSwapRef* 는 루프를 포함하지 않으며
따라서 행동에 관계없이 항상 진행할 것입니다
다른 동시 작업. 나머지 동시
여러 개의 잠재적으로 무한 루프가 있습니다
작업 종료를 지연시킬 수 있습니다. 우리는 전화
이 루프는 루프를 재 시도합니다. 우리가 조건을 생략하면
연산 의미론 때문에 (예 :
올바른 기준 등), 하위 작업시 루프 재시도
공유 변수의 값이 변경되었음을 감지하십시오. 이건
후속 읽기 하위 조작 또는 실패로 인해 tected
CAS. 이 공유 변수는 동시에 변경됩니다
다른 CAS 하위 작업에 의해. 줄에서 읽기 작업
성공적인 CAS 작업으로 인해 D5가 실패 할 수 있음
라인 C1, TN7 또는 CN5에서. 마찬가지로 CAS 작업
C1 라인에서 TN7 또는 CN5는
다른 CAS 작업이 성공했습니다. 에 따르면
동시 CAS 수에 대한 CAS 정의
하위 작업은 정확히 하나만 성공합니다. 이것은
이후 재 시도를 위해서는 하나의 CAS가 있어야합니다.
멈췄다. 이 후속 CAS는 재시도 루프를 발생시킵니다.

종료하고 구현에 순환이 포함되어 있지 않습니다.
CAS로 종료되는 재시도 루프 간의 종속성
해당 *DeRefLink* 작업 또는 *Ter-*
minateNode 하위 작업이 진행됩니다.

DeleteNode 작업 에는 세 개의 하위
운영, *CleanUpLocal* , *스캔* 및 *CleanUpAll* 하는
무한 루프 내부의 루프를 포함합니다. 루프
CleanUpLocal 에 의해 사용되는 하위 *조작* *CleanUpNode*
그리고 *CleanUpAll* 은 *합의* 가 없을 때 제한됩니다.
가정 1로 인해 *DeleteNode* 작업 임대
동시 *DeleteNode* 작업이 있습니다. *CleanUpN-*
송시 그들이 mm_del 비트를 설정하는 것이 단지 진행 상황, 즉
추가 노드에서 *CleanUpN-*
ODE 계속합니다. 따라서 *CleanUpNode* 는 잠금이 없습니다. 루프
에 *CleanUpAll* 모든 경계 루프에서 *CleanU-*
pLocal 과 *Scan* 의 크기는
DL_Nodes 목록은 정리 3에 의해 제한됩니다.
DeleteNode 는 정리 3의 경계에 의해 제한됩니다.

결과적으로, 동시 수에 관계없이
하나의 작업은 항상 진행됩니다. *2*

정리 5 *알고리즘은 잠금없는 가비지를 구현합니다.*
수집기.

증명 : Lemma 12는 구현에 잠금이 없음을 알려줍니다. *2*

6 실험 평가

우리는 추정 노력에 실험을 수행했습니다
새로운 잠금없는 메모리 사용의 평균 오버 헤드
이전 잠금이없는 것과 비교 한 관리 알고리즘
참조 카운트를 지원하는 메모리 관리 체계
노래. 이를 위해 잠금이없는 알고리즘을 선택했습니다.
deque (double-ended queue) 데이터 구조의 연산
Sundell과 Tsigas [20, 16]. 제시된 바와 같이, 구현 자들은
이 알고리즘의 tation은 lock-free 메모리 관리를 사용합니다.
Valois et al. [22, 13]. 또는
새로운 메모리 관리 체계에 더 잘 맞도록
deque 알고리즘의 재귀 호출이 풀 렸습니다.

우리의 실험에서 각 동시 스레드는 수행
공유에서 무작위로 선택된 10000 개의 순차적 작업
양단의 사이에 균등 분포 *PushRight* ,
PushLeft , *PopRight* 및 *PopLeft* 작업. 각 전
50 회 반복하고 평균 실행
각 실험에 대한 시간이 추정되었다. 정확히 동일
모든 다른 이미지에 대해 일련의 작업이 수행되었습니다.
plementation 비교.

실험은 다른 수를 사용하여 수행되었습니다
증가하는 단계에 따라 1에서 16까지 다양한 스레드 수.
실험에서 우리는 두 가지 구현을 비교합니다.
잠금이없는 데크; i) 잠금없는 메모리 사용
Valois 등의 노후화, 및 ii) 새로운 잠금없는 사용
메모리 관리 (동적 번호 지원 포함)
스레드 당 6 개의 위험 포인터가있는 스레드 수). 이들
(i) 앉은 유일한 메모리 관리 체계
lock-free deque 알고리즘의 요구를 충족시킵니다.
트래버스해야하는 다른 일반적인 잠금없는 알고리즘으로
동시에 삭제할 수있는 노드를 통해
그림의 예제에 사용 된 큐 알고리즘 8) 및
(ii) 이용 가능한 원자 프리미티브로 작업하십시오. 두 가지 구현
스테이션은 공유 고정 크기 메모리 풀 (예 : 프리리스트)을 사용합니다.
메모리 할당 및 해제 두 개의 다른 플랫폼
다양한 수의 프로세서와 레벨로
공유 메모리 분배. 먼저, 우리는 ex-
Linux를 실행하는 4 프로세서 Xeon PC의 성능. 또는
더 높은 동시성으로 알고리즘을 평가합니다.
8 프로세서 SGI Origin 2000 시스템도 사용
Irix 6.5. 클린 캐시 작업은
각각의 하위 실험. 모든 구현이 작성되었습니다
C에서 가장 높은 최적화 수준으로 컴파일됩니다. 그만큼
원자 기본 요소는 어셈블리로 작성됩니다. 결과
실험은 그림 9에 나와 있습니다. 평균 실행
tion time은 스레드 수의 함수로 그려집니다.

우리의 결과는 새로운 lock-free memory가
연령 알고리즘은 해당 알고리즘보다 성능이 뛰어납니다.
Valois et al. 스레드 수에 관계없이 광고
새로운 알고리즘을 사용하는 것의 장점은
비 균일 메모리 아키텍처를 가진 시스템에 중요
사실.

7 결론

우리가 아는 한, 첫 번째를 발표했습니다
잠금없는 가비지의 잠금없는 알고리즘 구현
모든 참조 카운트를 기반으로 수집 체계
다음과 같은 특징 : i) 다음과 같이 지역의 안전을 보장한다
전체 참조뿐만 아니라, ii)
삭제되었지만 아직 회수되지 않은 노드, iii)은
임의의 메모리 할당 체계, 그리고 iv) 원자를 사용
현대 건축에서 사용 가능한 기본 요소.

실험 결과에 따르면 새로운 잠금 장치가 없음
가비지 콜렉션은 성능을 크게 향상시킬 수 있습니다.
잠금없는 dy-
글로벌 참조의 안전을 요구하는 namic 데이터 구조
잘못되었습니다. 우리는 우리의 구현이
다중 프로세서 애플리케이션에 대한 실질적인 관심. 우리는
현재 그것을 NOBLE [19] 라이브러리에 통합하고 있습니다.

참고 문헌

[1] D. Detlefs, P. Martin, M. Moir 및 G. Steele Jr,“Lock-free
참조 카운트,” *20 연례 ACM의 절차에서*
분산 컴퓨팅 원리에 관한 심포지엄, 8 월
2001.

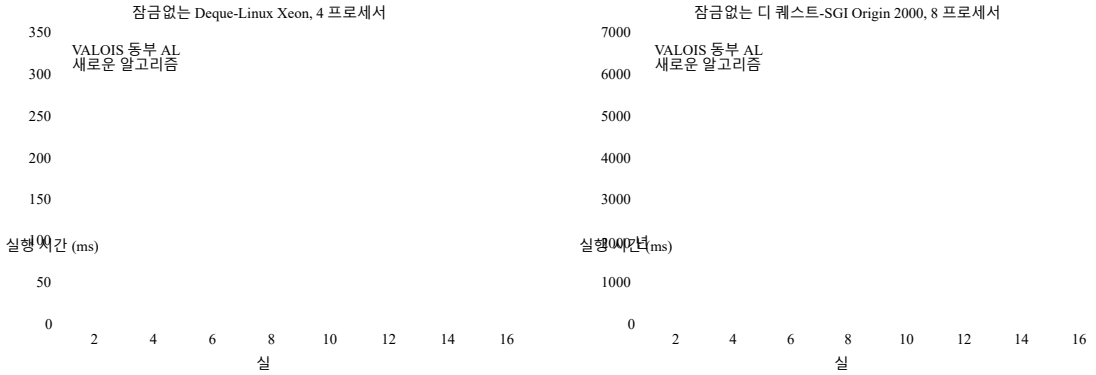

그림 9. 잠금없는 큐와 다양한 메모리 관리 체계를 실험하십시오.

[2] A. Gidenstam, M. Papatriantafilou 및 P. Tsigas,“Allocat-
잠금없는 방식으로 메모리를 사용하는 것”이라고 Computing Science,
찰머스 공과 대학교 Rep. 2004-04,
2004.

[3] M. Herlihy,“대기없는 동기화” *ACM 거래*
프로그래밍 언어 및 시스템, vol. 11 호 1, pp.
124–149, 1991 년 1 월.

[4] M. Herlihy, V. Luchangco, P. Martin 및 M. Moir,
Proceedings의"동적 크기의 잠금없는 데이터 구조"
불일치 원칙에 관한 제 21 회 연례 심포지엄
찬사 컴퓨팅. ACM Press, 2002, 131–131 쪽.

[5] M. Herlihy, V. Luchangco 및 M. Moir,“반복
펜더 문제 : 다이나믹을 지원하기위한 메커니즘
16 번째 Inter - Proceedings 에서 크기, 잠금이없는 데이터 구조”
분산 컴퓨팅에 관한 전국 심포지엄, 2002 년 10 월.

[6] M. Herlihy와 J. Wing,“Linearizability : 정확성
동시 객체에 대한 정보”, *ACM Transactions on Pro-*
문법 언어 및 시스템, vol. 12 번 3, 463 쪽 –
492, 1990.

[7] WH Hesselink 및 JF Groote,“Waitfree Distributed Mem-
삭제까지 생성 및 읽기 (CRUD)를 통한 ory 관리”
CWI, 암스테르담, 테크. 담당자 SEN-R9811, 1998.

[8] ——“만들어 대기없는 동시 메모리 관리
삭제까지 읽을 수 있습니다 (CaRuD).” *Distributed Computing* ,
vol. 14 번 1, pp. 31–39, 2001 년 1 월.

[9] P. Jayanti,“완전하고 일정한 시간 대기없는 간단한
DISC 1998 에서 ll / sc에서 cas를 언급하고 그 반대의 경우도 마찬가지입니다 .
1998, 216–230 쪽.

[10] MM Michael,“동적을위한 안전한 메모리 교정
잠금이없는 개체에서 "원자는 읽기 및 쓰기 사용 *Proceed-*
분산 원칙에 관한 제 21 회 ACM 심포지엄
컴퓨팅, 2002, 21–30 페이지.

[11] ——,“위험 포인터 : 잠금을위한 안전한 메모리 교정
무료 객체”” *병렬 및 분산에서의 IEEE 트랜잭션*
시스템, vol. 15 번 2004 년 8 월 8 일.

[12] ——“확장 가능한 잠금없는 동적 메모리 할당”
2004 ACM SIGPLAN 컨퍼런스 진행
프로그래밍 언어 설계 및 구현, 6 월
2004, pp. 35–46.

[13] MM Michael과 ML Scott,“기억의 교정
잠금없는 데이터 구조를위한 관리 방법”
로체스터 대학, 퍼터 과학부
1995 년.

[14] M. Moir,“비 차단 동기화의 실용적인 구현
에서 chronization 프리미티브 " *연간 15 논문집*
분산 컴퓨팅의 원리에 관한 ACM 심포지엄
ing , 1997 년 8 월.

[15] M. Moir, V. Luchangco 및 M. Herlihy,“Lock-free imple-
동적 크기의 공유 데이터 구조에 대한 언급”, Interna-
국제 특허 WO 2003/060 705 A3, 2003 년 1 월.

[16] H. Sundell,“효율적이고 실용적인 비 차단 데이터 구조
tures,”Ph.D. 찰머스 공과 대학교 논문
ogy, 2004 년 11 월

[17] ——,“대기없는 레퍼런스 카운팅 및 메모리 관리-
멘탈”, 찰머스 공과 대학교 컴퓨팅 과학
ogy, Tech. 2004 년 11 월, 2004 년 11 월.

[18] ——,“대기없는 레퍼런스 카운팅 및 메모리 관리-
” *, 제 19 회 국제 병렬 및*
분산 처리 심포지엄. IEEE 프레스, 2005 년 4 월

[19] H. Sundell 및 P. Tsigas,“고귀한 : 비 차단 인터
프로세스 커뮤니케이션 라이브러리” *, 6th Proceedings of 6th*
언어, 컴파일러 및 런타임 시스템에 대한 워크샵
확장 가능한 컴퓨터, ser. 컴퓨터 과학 강의 노트
그렇습니다. Springer Verlag, 2002.

[20] ——,“자유롭고 실용적인 이중 연결리스트 기반 장치
단일 단어를 사용 QUES 비교 및 스왑 "의 *절차*
제 8 차 국제 원칙 회의 국제 회의
찬사받은 시스템. 컴퓨터 과학 강의 노트,
Springer Verlag, 2004 년 12 월.

17 페이지

[21] JD 루아 "무 로크 큐를 구현"에 *Proceed-
*제 7 차 국제 회의 (Parallel and
분산 컴퓨팅 시스템, 1994, 64–69 페이지.

[22] —“Lock-free 데이터 구조”Ph.D. 논문, Rens-
Selaer Polytechnic Institute, 트로이, 뉴욕, 1995.