**Practical and Efficient Lock-Free Garbage Collection Based on Reference Counting**

**Anders Gidenstam Marina Papatriantafilou Håkan Sundell**

**Philippas Tsigas**

**Abstract**

우리는 임의의 lock-free 동적 데이터구조와 함께 사용하기위한 reference counting을 기반으로 garbage collection 체계의 효율적이고 실용적인 lock-free구현을 제시합니다. 이 체계는 로컬 및 글로벌 참조의 안전성을 보장하고, 임의의 메모리 재사용을 지원하며, 현대적인 컴퓨터 시스템에서 사용할 수 있는 원자적인 프리미티브를 사용하며 재사용이 금지된 메모리의 경계 상한을 제공합니다. (포인터 사이클?: provides an upper bound on the memory prevented for reuse.) 우리가 아는 한, 이 속성을 모두 제공하는 최초의 lock-free 알고리즘입니다. 실험 결과는 강력한 garbage collection이 필요한 lock-free 데이터구조의 성능이 크게 향상되었음을 나타냅니다.

Keywords: reference counting, garbage collection, lock-free, shared memory.

**1 Introduction**

동적 원자적 데이터구조를 구축하려면 메모리 관리가 필수적입니다. 데이터구조 및 관련 메모리 관리를 위한 원자적 알고리즘은 일반적으로 상호 배제를 기반으로 합니다. 그러나 상호배제(lock)는 blocking을 유발하므로 교착 상태(deadlocks), 우선 순위 반전(priority inversion) 또는 기아(starvation)와 같은 심각한 문제가 발생할 수 있습니다. 연구원들은 상호배제를 기반으로 하지 않는 non-blocking 동기화 알고리즘을 도입하여 이러한 문제를 해결했습니다. Lock-free 알고리즘은 non-blocking이며, 동시 작업에서 수행한 작업과 관계없이 항상 적어도 하나 이상의 작업이 진행될 수 있도록 합니다. Wait-free [3] 알고리즘은 lock-free이며, 동시 작업에서 수행한 작업에 관계없이, 모든 작업이 각자의 단계로 완료될 수 있습니다. Non-blocking 알고리즘에서는 동시 작업의 영향이 일관된 방식으로 뒤얽힌 프로세스에서 관찰될 수 있어야합니다. 일반적으로 일관성 요구 사항을 **선형화(linearizability)**[6]라고 합니다.

이 논문에서 우리는 lock-free 동적 데이터구조의 맥락에서 실용적이고 효율적인 메모리 관리에 중점을 두고 있습니다. 알고리즘의 동작이 lock-free가 되려면, 모든 서브 동작이 최소한 lock-free이어야 합니다. 결과적으로, lock-free 동적 데이터구조에는 일반적으로 lock-free메모리 관리가 필요합니다. 메모리 관리 문제는 일반적으로 동적 메모리 할당과 garbage collection의 하위 문제로 구분됩니다. Valois와 Michael과 Scott [22, 13]은 고정 크기 메모리 세그먼트를 위한 메모리 할당 체계를 제시했다. 이 체계는 상응하는 garbage collection체계와 함께 사용해야합니다. 일반적인 사용을 위한 Lock-free메모리 할당 체계는 Michael [12]과 Gidenstam et al에 의해 제시되었습니다. [2].

다양한 lock-free garbage collection 체계가 문헌에 제시되어 있다. Michael [10, 11]은 지역 참조에 중점을 둔 위험 포인터 알고리즘을 제안했다. 유사한 체계가 Herlihy et al.에 의해 제안되었다. [5] 이 체계는 제한이 없는 태그를 사용하며, 두 개의 인접한 메모리 워드를 원자적으로 업데이트 할 수 있는 비교 및 교환 작업 인 double-width CAS 원자 프리미티브를 기반으로 하며, 일부 32 비트 아키텍처에서 사용할 수 있지만 현재 64 비트 아키텍처 중 일부에서만 사용할 수 있습니다. 이러한 체계는 스레드에서 로컬 포인터의 안전성만 보장하므로 개체에 대한 전역 참조(즉 데이터구조 내의 포인터)를 항상 신뢰할 수 있어야하는 임의의 lock-free 알고리즘을 지원할 수 없습니다. 이 제약은 강하고 엄격하고, 동시성의 수준 증가 큰 성능 저하를 생성하여, 가능성이 큰 데이터 구조에 재시 순회하는 데이터 구조 알고리즘을 야기할 수 있다. (??? :and causes the data structure algorithms to retry traversals in the possibly large data structures, with resulting large performance penalties that increase with the level of concurrency.)

Reference counting에 기반한 garbage collection 체계는 객체에 대한 지역 참조 및 전역 참조의 안전을 보장할 수 있습니다. Valois et al. [22, 13]은 이용가능한 원자 프리미티브를 사용하여 구현할 수 있는 Lock-free reference counting체계를 제시했지만, 이는 메모리 할당을 위한 알고리즘과 사용하도록 제한되어 있다. Detlefs et al. [1]은 재생 메모리를 임의로 재사용 할 수 있는 체계를 제시했지만, 두 word의 임의 메모리 word를 원자적으로 업데이트 할 수 있는 double-words 비교 및 ​​교환 작업으로 현대에서는 사용할 수 없는 구조인, DCAS를 기반으로 한다. Herlihy et al. [4, 15]는 참조 계산 부분에 CAS만 사용하도록 이전 체계의 수정을 제시했다. 하지만, 이 체계는 자체에 double-width CAS가 필요한 다른 체계에 의존합니다. [13]에서 참조 카운트 기법은 잠재적으로 스레드로부터의 참조가 임의의 노드를 재할당하는 것을 차단(순환 참조를 생성하는 기능 때문에) 할 수 있음을 확인했다.

[double-width CAS : 두 개의 인접한 메모리 워드를 원자 적으로 업데이트 할 수 있는 비교 및 교환 작업은 일부 32 비트 아키텍처에서 사용할 수 있지만 현재 64 비트 아키텍처 중 일부에서만 사용할 수 있습니다.]

Wait-free 메모리 관리와 관련하여, Sundell이 Valois의 체계를 Wait-free 확장으로 제시했습니다 [18, 17]. Hesselink와 Groote [7, 8]는 토큰 공유의 특정 문제로 제한되는 Wait-free메모리 관리 체계를 제시했습니다.

이 논문은 각각의 단점을 피하면서 관련 기술의 장점을 유지하는 목적으로, 일반적인 lock-free reference counting 방식으로, 위험 포인터의 효율 참조 횟수의 강도를 결합합니다. 우리의 새로운 lock-free garbage collection기법은 lock-free이고 선형화(linearizable)이며, 메모리 할당에 대한 임의의 체계와 호환하고, 일반적으로 사용되는 원자 프리미티브를 사용하여 구현할 수 있으며 지역뿐만 아니라 전역 참조의 안전을 보장할 수 있습니다. 우리는 또한 일시적으로 스레드가 보유할 수 있는 메모리의 양을 결합하는지 보여줍니다.

논문은 다음과 같이 구성되어 있습니다. 2 장에서 우리는 우리의 구현이 목표로 하는 시스템의 유형을 설명합니다. 섹션 3은 우리가 초점을 맞추고 있는 garbage collection문제의 세부 사항을 설명합니다. 실제 알고리즘은 섹션 4에 설명되어 있습니다. 5장에서 우리는 우리의 구현에 대한 연산의 정확한 의미를 정의하고, lock-free와 선형성 특성을 증명할 뿐만 아니라 회수를 위해 일시적으로 보유할 수 있는 메모리의 경계를 증명해 알고리즘의 정확성을 보여줍니다. 6 장은 lock-free 데이터구조와 관련하여 새로운 체계의 실험적 평가를 제시한다. 우리는 7 장으로 논문을 마무리합니다.

**2 System Description**

공유 메모리 다중 프로세서 시스템 구성의 일반적인 추상화는 그림 1에 나와 있습니다. 시스템의 각 노드는 로컬 메모리와 함께 프로세서를 포함합니다. 모든 노드는 상호 연결 네트워크를 통해 공유 메모리에 연결됩니다. 각 작업을 수행하는 시스템에서 일련의 협력 작업이 실행 중입니다. 각 작업은 프로세서 중 하나에서 순차적으로 실행되는 반면 각 프로세서는 한 번에 많은 작업을 수행(실행)할 수 있습니다. 다른 프로세서에서 실행중인 협력 작업은 공유 메모리에 내장된 공유 데이터 객체를 사용하여 조정하고 통신합니다. 작업은 캐시 일관성 공유 메모리에서 서브 작업을 통해 공유 데이터 객체에 대한 작업을 동기화합니다. 그러나 공유 메모리는 시스템의 모든 노드에 대해 균일하게 액세스 할 수 없습니다. 프로세서는 메모리의 다른 부분에서 다른 액세스 시간을 가질 수 있습니다. 공유 메모리 시스템은 동기화를 위해 단일 메모리 단어의 원자적인 읽기 및 쓰기 작업뿐만 아니라, 강한 원자적 프리미티브를 지원해야합니다. 이 논문에서는 **FAA (Fetch-and-Add) 및 CAS (Compare-And-Swap)** 원자 프리미티브를 사용합니다. 설명은 그림 2를 참조하십시오. 이러한 읽기-수정-쓰기(read-modify-write) 스타일의 연산은 대부분의 일반적인 아키텍처에서 사용 가능하거나 다른 동기화 프리미티브에서 쉽게 파생될 수 있습니다 [14] [9].

**3 Problem Description**

이 논문에서는 동적 lock-free 데이터구조의 맥락에서 garbage collection 문제를 해결하는 것을 목표로 하고 있습니다. lock-free 데이터구조는 일반적으로 각각 임의의 데이터를 포함하는 **노드**라고하는 일련의 메모리 세그먼트로 구성됩니다. 이러한 노드는 임의 패턴으로 서로 참조하여 상호 연결됩니다. 참조는 일반적으로 메모리 주소를 통해 각 개별 노드를 식별할 수 있는 포인터를 사용하여 구현됩니다. 각 노드는 다른 노드를 참조하는, **링크**라 하는, 임의의 수의 포인터를 포함할 수 있습니다. 링크를 통해 참조된 노드를 따르는 작업을 **역참조**라고 합니다. 일부 노드는 일반적으로 항상 데이터구조의 일부이고, 다른 모든 노드는 데이터구조의 일부인 노드가 참조할 때 데이터 구조의 일부입니다. 동적 및 동시성 데이터구조에서 임의의 노드는 데이터구조에서 지속적으로 동시에 추가되거나 제거될 수 있습니다. 시스템의 메모리 양이 제한되어 있으므로, 이러한 노드의 점유된 메모리는 시스템에서 동적으로 할당 및 재생되어야 합니다. 데이터 구조의 순차적 인 구현에서, 노드의 메모리는 일반적으로 노드에 대한 마지막 참조가 제거될 때, 즉 노드가 삭제될 때 시스템에 명시적으로 재생된다. 동시 환경에서는 재사용된 노드의 메모리에 대한 접근 가능성이 데이터구조 및/또는 전체 시스템의 정확성에 치명적이기 때문에 스레드에 있을 수 있는 노드에 대한 가능한 로컬 참조도 포함해야 합니다. (공유메모리 가리키는 지역 포인터?) 재사용을 올바르게 결정하는 논리 단위를 **garbage collector**라고 하며 다음 특성이 있어야 합니다.

**Property 1** Garbage collector는 데이터구조의 일부가 아니며 스레드에 의해 향후에 접근이 불가능한 garbage만 회수해야 합니다.

데이터구조에 사용되는 최대 메모리 양을 항상 예측할 수 있어야 하므로, garbage collector에 이 요구 사항을 추가할 수 있습니다.

**Property 2** 언제라도, 데이터 구조의 일부는 아니지만 아직 시스템으로 재사용되지 않은 노드 수에는 상한이 있어야합니다. (포인터 사이클?)

Garbage collector (GC)의 실제 구현에서 이러한 속성은 노드에 대한 로컬 참조가 전역으로 접근할 수 없으므로 (예: 프로세서 레지스터에 저장될 수 있음) 달성하기가 매우 어려울 수 있습니다. 따라서 GC 구현은 일반적으로 관련 스레드와 상호 작용하고 노드에 대한 접근에 제한을 두어야 합니다. 예: 링크 역참조와 노드가 삭제될 때 데이터 구조의 구현이 명시적으로 garbage collector를 호출할 것을 요구하기위한 특수 동작을 제공함으로써.

또한, 관심있는 기본 데이터 구조는 lock-free이고 일반적으로 선형화 가능하므로, garbage collector는 다음 기능을 보장해야 합니다.

**Property 3** 기본 데이터 구조 구현과 통신하기위한 garbage collector의 모든 동작은 lock-free이고 선형화 되어야합니다.

다양한 데이터 구조에 대해 전체 시스템의 총 점유 메모리를 최소화하기 위해 언젠가 다음 속성을 달성해야 합니다.

**Property 4** garbage collector에서 회수한 메모리는 향후 임의의 재사용을 위해 접근할 수 있어야합니다. 즉, garbage collector는 시스템의 기본 메모리 할당자와 호환되어야 합니다.

동시 환경에서 스레드가 다른 스레드에 의해 삭제된 (즉, 데이터 구조에서 제거된) 노드에 대한 로컬 참조를 보유하는 경우가 종종 있습니다. 이 경우 첫 번째 스레드가 삭제된 노드의 링크를 사용할 수 있으면 매우 유용할 수 있습니다. 예: 큰 데이터 구조의 검색 절차에서.

**Property 5** 노드에 대한 로컬 참조가 있는 스레드는 해당 노드에 포함된 모든 링크를 역참조를 할 수 있어야합니다.

이 논문의 새로운 알고리즘은 현대 시스템에서 일반적으로 사용 가능한 원자 프리미티브만 사용하는 특성 외에도 이러한 모든 특성을 충족합니다. 표 1은 이전에 제시된 lock-free garbage collector 체계와 달성한 특성의 비교를 보여줍니다. 모든 체계는 특성 1과 3을 충족하지만, 반면에 다른 특성들 중 일부만 이전에 제시된 체계에 의해 충족된다.

**4 The New Lock-Free Algorithm**

섹션 3에서 요구된 속성을 모두 만족할 뿐만 아니라 효율적이고 실용적인 방법을 제공하기 위해서, 우리의 목표는 Michael의 위험 포인터 (HP) 체계를 사용할 수 있는 reference counting 방법을 고안하는 것입니다. [10, 11] 대략적으로 말하면, 위험 포인터는 4개의 데이터 구조에서 내부 링크의 안전을 보장하기 위해 로컬 참조 및 reference count의 안전을 보장하는 데 사용됩니다. 따라서, 각 노드의 참조 수는 해당 노드를 참조하는 접근가능한 전역 링크 수를 나타냅니다. 그림 3은 알고리즘에서 사용되는 노드 구조를 설명합니다. HP 체계와 같이, 각 스레드는 삭제되었지만 아직 재사용되지 않은 노드 목록을 유지 관리하며, 이 목록의 길이가 특정 임계 값에 도달하면 재사용이 가능한지 검사합니다. (예: THRESHOLD\_2) 삭제된 노드 중 일부는 고정된 수의 위험 포인터로 인해 재사용되지 않을 수 있는 반면, 일부 삭제된 노드는 링크에 대한 양의 reference count로 인해 방지될 수 있습니다. 따라서, 링크에서 삭제된 노드에 대한 참조수를 최소로 유지하는 것이 중요합니다. 삭제 목록의 크기를 제한하는 기술을 계속하기 전에 lock-free 데이터 구조 알고리즘에 필요한 사항에 대한 가정을 소개합니다.

**Assumption 1** 삭제된 노드를 참조하는 삭제된 노드의 각 링크에 대해, 관련 스레드에 대한 의미가 유지되는, 활성 노드에 대한 참조로 대체할 수 있어야합니다.

이 가정의 직관은 삭제된 노드의 링크가 로컬 참조가 있는 스레드에서 역참조를 하는 유용한 이유를 관찰한 것입니다. 삭제된 노드에 대한 로컬 참조가 있는 스레드는 반드시 적절한 활성 노드를 찾고 그래서 링크를 이용합니다. 해당 참조도 삭제된 노드를 준수하면 이전 단계가 반복됩니다. 관심있는 스레드의 관점에서 볼 때, 다른 스레드가 프로시저에 도움이 되고 삭제된 노드의 링크가 모두 활성 노드를 참조하는지 확인하면 아무런 차이가 없습니다. 삭제된 노드의 링크를 활성 노드에 대한 참조로 바꾸는 절차를 **정리**라고 합니다.

앞에서 설명한 것처럼, 위험 포인터 외에도, 삭제목록의 노드는 다른 삭제된 노드의 링크에 의해 재사용되지 않을 수 있습니다. 이러한 노드는 동일한 삭제 목록 또는 다른 스레드의 삭제 목록에 있을 수 있습니다. 이러한 이유로 모든 스레드의 모든 삭제 목록을 읽을 수 있습니다. 삭제 목록의 길이가 특정 임계 값 (THRESHOLD\_1)에 도달하면 스레드는 삭제 목록의 모든 노드를 정리합니다. 모든 노드가 여전히 재사용되지 않으면, 이는 다른 스레드의 삭제목록에 있는 노드로 인한 것이므로, 스레드는 다른 모든 스레드의 삭제 목록도 정리하려고 합니다. 삭제목록의 길이가 임계 값 미만이 될 때까지 이 절차가 반복되므로, 아직 재사용되지 않은 삭제된 노드의 양이 제한됩니다. THRESHOLD\_1의 실제 계산은 부록 5.2에 설명되어 있습니다. 임계 값 THRESHOLD\_2는 HP 체계에 따라 설정되거나 THRESHOLD\_1보다 작거나 같습니다.

**4.1 Application Programming Interface**

참조 카운트 노드를 안전하게 처리하기 위해 다음 기능이 정의됩니다.

**DeRefLink** 함수는 주어진 링크를 안전하게 참조 해제하고, 참조 해제된 노드에 대한 위험 포인터를 설정하여 리턴 된 노드에 접근할 수 있도록 향후 안전을 보장합니다. 주어진 노드에 더 이상 현재 스레드가 접근하지 않을 때 **ReleaseRef** 함수를 호출해야 합니다. 해당 위험 포인터가 지워집니다.

링크에 대한 동시 업데이트를 하려면 링크를 업데이트하기 위해서, **CompareAndSwapRef** 함수를 사용해야 하고, 업데이트 성공여부 결과가 표시됩니다. 스레드에 링크가 포함된 노드에 대한 위험 포인터 참조가 있는 경우, 함수는 링크에서 DeRefLink를 호출하는 모든 스레드가 안전하게 그렇게 할 수 있는지 확인합니다. 요구 사항은 CompareAndSwapRef의 호출 스레드에 지정된 노드에 대한 위험 포인터가 있어야 한다는 것입니다.

동시 업데이트가 없는 링크를 업데이트하기 위해서 **StoreRef** 함수를 호출해야 합니다. 스레드에 링크가 포함된 노드에 대한 위험 포인터 참조가 있는 경우, 함수는 링크에서 DeRefLink를 호출하는 스레드가 안전하게 그렇게 할 수 있는지 확인합니다. 요구 사항은 StoreRef의 호출 스레드에 지정된 노드에 대한 위험 포인터가 있어야 하며 다른 스레드가 링크에 동시에 쓰지 않아야 한다는 것입니다 (그렇지 않으면 CompareAndSwapRef가 대신 호출되어야 함).

**NewNode** 함수는 새로운 노드를 할당하고, 미래의 접근 안전을 보장하기 위해 자유 위험 포인터를 설정한 다음, 이것을 리턴합니다. **DeleteNode** 함수는 노드가 데이터 구조에서 제거되고 재사용 할 수 있는 메모리가 필요할 때 호출되어야 합니다. DeleteNode라는 사용자 작업은 데이터 구조의 활성 노드에서 삭제된 노드에 대한 모든 참조를 제거합니다. 이는 순차 데이터 구조에서 메모리 할당자를 사용할 때 필요한 것과 유사합니다. 메모리 관리자는 안전하게 삭제할 때까지 삭제된 노드를 회수하지 않습니다. 4.4 장에서 우리는 이 함수들이 어떻게 연결리스트를 기반으로 하는 lock-free큐 알고리즘의 맥락에서 사용될 수 있는지에 대한 예를 제공한다.

**Callbacks**

다음 함수는 각 특정 데이터 구조의 디자이너가 정의해야 하는 callback입니다. **TerminateNode** 함수는 지정된 노드의 링크 중 다른 노드에 대한 소유권이 없는지 확인합니다. TerminateNode는 다른 노드 또는 스레드에서 해당 노드로 소유권이 없는 경우 삭제된 노드에서 호출됩니다.

[원칙적으로 이 절차는 메모리 관리자가 제공할 수 있지만 실제로 사용자가 노드 레코드의 메모리 레이아웃을 결정하도록 하는 것이 더 편리합니다. 모든 노드 레코드는 여전히 mm\_ref, mm\_trace 및 mm\_del 필드로 시작해야합니다.]

**CleanUpNode** 함수는 주어진 노드의 링크에서 요구된 모든 참조가 활성 노드만 가리키게 하여 임의의 수의 삭제된 노드를 통한(???) 중복 통로를 제거합니다.

**4.2 Auxiliary Procedures : 보조절차**

Reference counting 체계에 의해 내부용으로 정의된 보조 함수 :

**Scan** 함수는 이 스레드에 의해 삭제된 아직 재사용되지 않은 모든 노드를 검색하고 일치하는 위험 포인터가 없고 노드 내부의 링크에서 계산된 참조가 없는 노드만 재사용합니다. **CleanUpLocal** 함수는 이 스레드에 의해 삭제된 노드의 링크에서 중복된 소유권 참조를 제거하려고 시도합니다. **CleanUpAll** 함수는 스레드에 의해 삭제된 노드의 링크에서 중복된 소유권 참조를 제거하려고 시도합니다.

**4.3 Detailed Algorithm Description**

**DeRefLink** (그림 4)는 먼저 라인 D3에서 \* link에 저장된 노드에 대한 포인터를 읽습니다. 그런 다음 D4 라인에서 스레드의 위험 포인터 중 하나가 노드를 가리 키도록 설정합니다. D5 행에서 링크가 여전히 이전과 동일한 노드를 가리키는 지 확인합니다. \* link가 여전히 노드를 가리키는 경우, 노드가 아직 재사용되지 않았으며 이제 위험 포인터가 노드를 가리키고 있을 때까지 노드를 재사용할 수 없음을 알고 있습니다. 마지막으로 읽은 이후 \* link가 변경되면 다시 시도합니다.

**ReleaseRef** (그림 4)는 노드를 가리키는 이 스레드의 위험 포인터를 제거합니다. 노드가 삭제되고, reference count가 0이고 다른 위험 포인터가 이를 가리 키지 않으면, Scan함수(노드에서 DeleteNode라고하는 스레드에 의해 호출됨)에 의해 노드를 회수할 수 있습니다.

**CompareAndSwapRef** (그림 4)는 링크에서 공통 CAS를 수행하고 그에 따라 각 노드의 참조 횟수를 적절히 업데이트합니다. C4 라인은 동시성 Scan함수에 node의 참조 횟수가 증가했음을 알립니다. CompareAndSwapRef를 호출하는 스레드가 이를 가리키는 위험 포인터를 가져야하기 때문에 CompareAndSwapRef중에 노드에 접근하는 것은 안전합니다. 라인 C5에서, old의 참조 카운트가 감소된다. \* link가 이전 노드를 참조했으므로 old 참조 계수는 0보다 커야 합니다.

**StoreRef** (그림 4)는 \* link의 동시 업데이트가 없는 경우에만 사용할 수 있습니다. 라인 S2에서 \* link를 업데이트 한 후 StoreRef는 라인 S4에서 노드의 참조 카운트를 증가시킵니다. 이는 StoreRef를 호출하는 스레드가 노드에 대한 위험 포인터를 가져야하기 때문에 안전합니다. 라인 S5는 모든 동시 스캔에 노드의 참조 카운트가 0이 아님을 알립니다. 라인 S6에서, 이전의 참조 카운트가 감소된다. \* link가 이전 노드를 참조했으므로 이전 참조 계수는 0보다 커야 합니다.

**NewNode** (그림 4)는 기본 메모리 할당자로부터 새 노드를 위한 메모리를 할당하고 각 노드가 가져야하는 헤더 필드를 초기화합니다. 또한 노드에 대한 위험 포인터를 설정합니다.

**DeleteNode** (그림 5)는 DN2 행에서 노드가 논리적으로 삭제된 것으로 표시합니다. 그런 다음 DN3-DN7 행에서 노드는 이 스레드에서 삭제되었지만 아직 회수되지 않은 노드 세트에 삽입됩니다. DN5 행에서 포인터를 쓰기 전에 DN4 행에서 DL\_Done을 정리함으로써 동시 CleanUpAll 작업이 노드에 접근하여 해당 참조를 정리할 수 있습니다.

이 스레드에서 삭제되었지만 아직 재사용되지 않은 노드 세트의 삭제된 노드 수가 THRESHOLD\_2보다 크거나 같은 경우 다른 노드 또는 스레드가 참조하지 않는 세트의 모든 노드를 회수하는 **Scan**이 수행됩니다.

스레드에서 삭제되었지만 아직 수정되지 않은 노드 세트가 이제 가득 찼다면, 즉, THRESHOLD\_1 노드 포함, 스레드는 먼저 DN9 행에서 **CleanUpLocal**을 실행하여 삭제된 모든 노드가 **CleanUpLocal**가 시작할 때 활성상태인 노드만 가리키도록 합니다. 그런 다음 DN10 라인에서 **Scan**을 실행합니다. 스캔이 노드를 전혀 회수할 수 없는 경우 스레드는 **CleanUpAll**을 실행하여 모든 스레드의 삭제된 노드 세트를 정리합니다.

**Callbacks**

**TerminateNode**(그림 6)는 노드의 링크에 NULL을 작성하여 노드의 모든 링크를 지워야 합니다. 이러한 링크의 동시 업데이트가 있는지 여부에 따라 CompareAndSwapRef 또는 StoreRef를 사용하여 수행됩니다.

**CleanUpNode** (그림 6)는 이 CleanUpNode호출이 시작되기 전에 삭제된 노드를 가리키는 노드 링크가 없는지 확인해야합니다.

**Auxiliary procedures**

**Scan** (그림 7)은 다른 노드 또는 위험 포인터가 참조하지 않는 현재 스레드에 의해 삭제된 모든 노드를 재생합니다. 안전하게 재생할 수 있는 삭제된 노드를 판별하려면 먼저 스캔이 참조 카운트가 0 인 모든 삭제된 노드의 mm\_trace 비트를 설정하십시오 (SC1-SC7 행). SC6행에서의 점검은 mm\_trace 비트가 설정될 때 기준 카운트가 실제로 0임을 보장한다.

그런 다음 Scan은 모든 스레드의 모든 활성 위험 포인터를 plist에 기록합니다. (SC8 ~ SC14 행) SC15-SC30행에서 Scan은 이 스레드에 의해 삭제된 아직 재생되지 않은 모든 노드를 통과합니다. 이러한 각 노드에 대해 SC19 라인의 테스트에서 i) 참조 카운트가 0인지, ii) 위험 포인터를 읽기 전 (mm\_trace 비트가 설정됨으로 표시) 이래로 참조 카운트는 지속적으로 0인지, iii) 위험 포인터가 노드를 참조하지 않는지 결정합니다. 이 세 가지 조건이 모두 해당한다면, 노드가 참조되지 않고 Scan은 SC21 행에서 노드에 대해 동시에 CleanUpAll 작업이 수행되고 있는지 확인합니다. 그러한 CleanUpAll 작업이 없는 경우 Scan은 TerminateNode를 사용하여 노드에 포함된 모든 참조를 해제한 다음 노드를 회수합니다 (SC22 및 SC23 행). 노드에 액세스하는 CleanUpAll 작업이 동시에 있을 수 있는 경우 Scan은 동시 버전의 TerminateNode를 사용하여 모든 노드의 링크를 NULL로 설정합니다. SC27 행에서 노드를 재사용되지 않은 노드 세트에 다시 삽입하기 전에 SC26 행에서 DL\_Done 플래그를 설정하면 CleanUpAll 조작으로 인해 이 노드가 후속 Scan에서 재사용되지 않을 수 없습니다. CleanUpLocal (그림 7)은 스레드의 삭제되었지만 재사용되지 않은 노드 목록을 탐색하고 각 노드에서 CleanUpNode를 호출하여 CleanUpLocal이 시작될 때 이미 삭제된 노드를 링크가 참조하지 않도록 합니다.

CleanUpAll (그림 7)은 모든 스레드의 DL\_Nodes 배열을 탐색하고 찾은 노드 중 CleanUpAll이 시작될 때 이미 삭제된 노드에 대한 링크가 없는 노드가 없는지 확인합니다. CA4 라인의 테스트는 CleanUpAll이 참조가 없는 노드를 위한 Scan을 불필요하게 방해하지 않도록 합니다. CA6 라인의 테스트는 CleanUpAll이 Scan이 이미 재사용한 노드에 접근하지 못하게 합니다. CA6 라인에서 노드가 DL\_Nodes [thread] [index]에 여전히 존재하는 경우이 노드에 접근하는 동시 스캔은 노드를 회수하지 않고 SC20 라인 이전 또는 SC27 라인 이후에 있어야합니다.

**4.4 Example Application**

동적 데이터 구조를 위한 lock-free알고리즘에 메모리 관리를 위한 새로운 알고리즘을 적용하는 것은 이전에 제시된 메모리 관리 체계와 유사한 방식으로 똑바로 수행될 수 있다. 그림 8은 Valois의 lock-free Queue 알고리즘을 보여줍니다. [21, 13]은 메모리 관리를 위한 새로운 알고리즘과 통합될 것이다.

**4.5 Algorithm Extensions**

단순화를 위해, 이 논문의 알고리즘은 고정 개수의 스레드로 설명됩니다. 그러나이 체계는 [11]의 HP 체계에 대해 설명된 것과 유사한 방식으로 동적 스레드 수에 대해 쉽게 확장할 수 있습니다. 위험 포인터(HP)의 전역 모형은 배열의 연결 리스트로 전환할 수 있습니다. 삭제목록은 전역 체인에 연결될 수도 있으며, 삭제 목록의 크기가 변경되면, 메모리 관리를 위해 추가 HP 체계를 사용하여 이전 중복 삭제 목록을 안전하게 회수할 수 있습니다.

**4.6 Algorithm Correctness and Bounds on Unreclaimed Memory**

**Theorem 1** 이 알고리즘은 garbage collection을 위해 lock-free하고 선형화 가능한 체계를 구현합니다.

**Theorem 2** 시스템에서 삭제되었지만 아직 재사용되지 않은 노드 수는 N^2 · (k + lmax + α + 1) 여기서 N은 시스템의 스레드 수, k는 스레드 당 위험 포인터 수, lmax는 노드가 포함할 수 있는 최대 링크 수, α는 삭제된 노드를 일시적으로 가리킬 수 있는 활성 노드의 최대 링크 수입니다.

위의 정리에 해당하는 증명은 섹션 5에 남겨져 있으며, 여기서 보조정리를 사용하여 제시됩니다.

**5 Correctness Proof**

**5.1 Safety**

**5.2 Proof of the bound on the number of deleted but unreclaimed nodes**

**5.3 Linearizability**

**5.4 Proof of the Lock-free Property**

**6 Experimental Evaluation**

우리는 참조 카운트를 지원하는 이전의 lock-free 메모리 관리 체계와 비교하여 새로운 lock-free 메모리 관리 알고리즘을 사용하는 평균 오버 헤드를 추정하기 위해 실험을 수행했습니다. 이를 위해 Sundell과 Tsigas에 의해 deque (double-ended queue) 데이터구조의 lock-free알고리즘을 선택했습니다 [20, 16]. 제시된 것과 같이, 이 알고리즘의 구현은 Valois et al.의 reference counting과 함께 lock-free 메모리 관리를 사용합니다. [22, 13]. 새로운 메모리 관리 체계에 더 잘 맞추기 위해 deque 알고리즘의 재귀 호출이 풀렸습니다.

우리의 실험에서, 각각의 원자적 스레드는 PushRight, PushLeft, PopRight 및 PopLeft 동작 중 동일한 분포로 공유 deque에서 무작위로 선택된 10,000개의 임의로 선택된 연속 작업을 수행하였습니다. 각 실험은 50 회 반복하였으며, 각 실험의 평균 실행 시간을 측정하였습니다. 모든 다른 구현 비교에 동일한 작업 시퀀스가 수행되었습니다.

단계가 증가함에 따라 1에서 16까지의 다른 스레드 수를 사용하여 실험을 수행했습니다. 실험에서 우리는 lock-free deque의 두 가지 구현을 비교합니다.

i) Valois의 lock-free 메모리 관리 사용

ii) 스레드 당 6 개의 위험 포인터가 있는 새로운 lock-free 메모리 관리 사용.

(동적 스레드 수 지원 포함)

이것들은 (i) lock-free deque 알고리즘 (뿐만 아니라 그림 8의 예에 사용된 Queue 알고리즘과 같이 동시에 삭제될 수 있는 노드를 통과해야 하는 다른 일반적인 lock-free 알고리즘)의 요구를 충족시키고 (ii) 사용 가능한 원자 프리미티브와 작동하는 유일한 메모리 관리 체계입니다. 두 구현 모두 메모리 할당 및 해제를 위해 공유 고정 크기 메모리 풀 (예: free-list)을 사용합니다. 다양한 수의 프로세서와 공유 메모리 분배 레벨을 가진 두 개의 서로 다른 플랫폼이 사용되었습니다. 먼저 Linux를 실행하는 4 프로세서 Xeon PC에서 실험을 수행했습니다. 동시성이 높은 알고리즘을 평가하기 위해 Irix 6.5를 실행하는 8 프로세서 SGI Origin 2000 시스템도 사용했습니다. Clean-cache 작업은 각 서브 실험 직전에 수행되었습니다. 모든 구현은 C로 작성되고 가장 높은 최적화 수준으로 컴파일 됩니다. 원자 프리미티브는 어셈블리로 작성됩니다. 실험 결과는 그림 9에 나와 있습니다. 평균 실행 시간은 스레드 수의 함수로 표시됩니다.

우리의 결과는 새로운 lock-free메모리 관리 알고리즘이 임의의 수의 스레드에 대해 Valois의 알고리즘을 능가함을 보여줍니다. 새로운 알고리즘 사용의 장점은 불균일(non-uniform) 메모리 아키텍처를 가진 시스템에서 훨씬 더 중요하다는 것을 보여줍니다.

**7 Conclusions**

우리가 아는 한, 우리는 다음과 같은 기능을 모두 갖춘 reference counting를 기반으로 lock-free garbage collection 체계의 최초의 lock-free 알고리즘 구현을 제시했습니다.

i) 지역뿐만 아니라 전역 참조의 안전을 보장합니다.

ii) 삭제되었지만 아직 회수되지 않은 노드의 바인딩을 제공합니다.

iii) 임의의 메모리 할당 체계와 호환됩니다.

iv) 현대 아키텍쳐에서 이용 가능한 원자 프리미티브를 사용한다.

실험 결과는 우리의 새로운 lock-free garbage collection이 전역 참조의 안전을 요구하는 lock-free동적 데이터구조 구현의 성능 및 안정성을 크게 향상시킬 수 있음을 나타냅니다. 우리는 우리의 구현이 다중 프로세서 응용 프로그램에 매우 실용적인 관심사라고 생각합니다. 우리는 현재 이것을 NOBLE [19] 라이브러리에 통합하고 있습니다.

**References**