**Lock-free reference counting**

**David L. Detlefs, Paul A. Martin, Mark Moir, Guy L. Steele Jr.**

Sun Microsystem Laboratories, 1 Network Drive, Burlington, MA 01803, USA

(e-mail: [Mark.Moir@sun.com](mailto:Mark.Moir@sun.com)) Received: January 2002 / Accepted: March 2002

**Abstract**

garbage collection이 있다고 가정하면 동적 크기의 공유 자료구조 구현을 보다 쉽게 설계할 수 있습니다. 그러나 이 가정은 적용 가능성을 제한합니다. 중요한 클래스의 자료구조에 대해 디자이너가 먼저 garbage collection 종속 구현 설계의 보다 쉬운 문제를 해결한 다음이 방법론을 적용하여 garbage collection 독립 구현 방식을 구현할 수 있는 방법론을 제시합니다. 우리의 방법론은 잘 알려진 reference counting 기술을 기반으로 하며, double compare-and-swap(DCAS) 작업을 사용합니다.

**Keywords**

Lock-free synchronization – Reference counting – Memory management – Dynamic data structures

**Introduction**

동적 크기의 nonblocking 공유 자료구조의 설계를 단순화하는 데 사용할 수 있는 방법을 제시합니다. 최근 몇 년 동안 nonblocking 자료구조의 구현에 지불된 상당한 연구 관심이 lock-free & wait-free 구현에 중점을 두었습니다. 연산의 유한 단계 수 이후에, (다른 작업을 현재 실행중인 스레드의 타이밍 동작에 관계없이) 작업이 완료되도록 보장하는 경우의 구현은 wait-free입니다. 한정된 수의 작업 단계 후에, 일부 작업이 완료되도록 보장하는 경우의 구현은 lock-free입니다. 잠금을 유지하는 동안 일부 스레드가 지연되면 -잠금이 필요한 단계가 많더라도- 잠금 홀더가 잠금을 해제할 때까지, 잠금이 필요한 다른 작업을 완료할 수 없기 때문에, 두 속성 모두 동기화에 상호 배타적 잠금을 사용할 수 없습니다. Lock-free 프로그래밍은 성능 병목 현상, 지연 및 장애에 대한 취약성, 설계 복잡 및 실시간 시스템의 우선 순위 반전을 포함하여, 잠금과 관련된 다른 문제를 극복하는 데 점점 중요 해지고 있습니다.

 Lock-freedom는 오직 하나의 시스템 진행을 보장하고, Wait-freedom은 작업 별 진행을 보장하므로, Wait-freedom보다 약합니다. 그러나 연구 경험에 따르면, Wait-freedom 속성이 강할수록 달성하기가 훨씬 어려우며 일반적으로 구현 비용이 더 많이 듭니다. 또한 잘 설계된 시스템에서는 경쟁이 낮아야 하기 때문에 실제로 Lock-freedom이 충분할 수 있습니다. 이 논문에서, lock-free 자료구조, 특히 동적화 된 구조, 즉 시간이 지남에 따라 크기가 커지고 축소될 수 있는 자료구조의 설계에 중점을 둡니다.

 garbage collection(GC)은 동적 크기 자료구조의 순차적 구현의 설계를 단순화할 수 있다는 것이 잘 알려져 있다. 또한 GC는 스토리지 관리 이점을 제공할 뿐만 아니라, 경쟁 자료구조 설계에서 다양한 동기화 문제를 크게 단순화할 수 있습니다 [3,15,22]. 그러나 불행하게도 garbage collection 환경에 대한 구현을 설계하는 것이 전체 솔루션이 아닙니다. 첫째, 모든 프로그래밍 환경이 GC를 지원하는 것은 아닙니다. 둘째, 잠금과 정지 메커니즘과 같이 과도한 동기화를 사용하는 거의 모든 것이, 확장성에 의문을 제기합니다. 마지막으로, 우리 그룹에 대한 특별한 관심의 명백한 제한점은 GC를 의존하는 구현이 garbage collector 자체 구현에 사용될 수 없다는 것입니다!

 따라서 이 작업의 목표는, 프로그래머가 lock-free 자료구조 구현을 개발하면서 GC 단점을 피하면서, GC의 장점을 활용할 수 있도록 하는 것입니다. 이를 위해, 프로그래머가 GC 의존적인 구현 설계의 보다 쉬운 문제를 먼저 해결한 다음, GC 독립적 구현을 ​​달성하기 위해 방법론을 적용할 수 있는 방법론을 제공합니다.

 lock-freedom를 유지하기 위해 방법론을 설계했습니다. 즉, GC 종속 구현이 lock-free라면, 방법론을 사용하여 파생된 GC 독립적 구현이 될 것입니다. 여기에 두 가지 설명이 필요합니다. 첫째, 일부 독자는 대부분의 garbage collector가 작업을 수행하는 동안 스레드의 실행을 중지한다는 점에서 lock-free GC 종속적인 구현에 대한 우리의 reference에 대해 우려할 수 있습니다. 그러나 이것이 자료구조 구현 자체가 lock-free가 아님을 의미하는 것은 아닙니다. lock-freedom의 정의는 시스템 (예: GC 또는 OS)으로 인해 스레드가 단계를 수행하지 못하게 하는 경우의 진행이 요구하지 않습니다.

 둘째, 방법론을 적용한 동적 크기 자료구조의 lock-free GC 독립적 구현을 ​​달성을 말하지, 객체 생성 및 소멸 방법은 지정하지 않습니다. 일반적으로 malloc 및 free는 lock-free이 아니므로 이를 기반으로 한 구현도 lock-free가 아닙니다. 그러나 대부분의 프로덕션 품질 메모리 할당자는 -각 스레드에 대해 별도의 할당 버퍼를 유지하는 등- 잠금 경합을 피하려고 시도하므로, 잠금과 관련된 대부분의 문제를 피합니다. 실제로 우리의 동료 인 Dice & Garthwaite는 최근에 지역성을 개선하고 동기화 오버헤드를 줄이기 위해 "멀티 프로세서 인식"으로 설계된 새로운 메모리 할당 기 [5]를 제안했습니다. 결과적으로 할당자는 잠금에만 매우 가끔 의존하며, 확장되므로 확장성이 매우 뛰어납니다. 할당자는 여전히 잠금을 사용하지만, 변형에 간단하고 잘 알려진 nonblocking기술의 변형을 적용하여 완전히 nonblocking 할 수 있습니다.

우리 방법론의 중요한 특징-이를 기반으로 하는 자료구조 구현-은 기본 메모리 할당 메커니즘에 제한을 두지 않고, 구현의 메모리 소비가 시간이 지남에 따라 증가 및 축소될 수 있다는 것입니다. 반대로, 동적 자료구조의 lock-free 구현은, 일반적으로 저장소를 다른 목적(예: [19,25])으로 재사용 할 수 없는 특수 "free-list"의 유지 보수가 필요하거나, 안정 메모리 유형과 같은 특수 시스템 지원이 필요합니다. 메모리 [7].

 우리의 방법론은 잘 알려진 참조 계산의 garbage collection 기술을 기반으로 합니다. 우리는 방법론을 LFRC (Lock-Free Reference Counting)라고 합니다. 이 방법에서 각 객체에는 해당 객체를 가리키는 포인터 개수가 포함되며 이 개수가 0 인 경우에만 해제할 수 있습니다. 메모리 관리에 대한 reference counting 접근 방식은 garbage의 사이클을 감지하지 못하는 한계가 있습니다 (각각 다음에 대한 포인터가 있는 도달할 수 없는 객체의 링을 상상하십시오. 모든 참조 카운트는 적어도 하나이므로 객체가 없습니다) 쓰레기로 식별). LFRC는 이 제한을 상속합니다. 대부분의 경우, 사이클이 garbage에 존재하지 않거나, 구현을 쉽게 수정하여 이 가능성을 제거할 수 있습니다. 다른 경우에는, 그렇게 하기가 더 어렵거나 불가능합니다. 이러한 경우 당사의 방법론은 모든 garbage가 반환될 것이라고 보장할 수 없습니다.

 정확한 reference counts를 유지하기 위해, 원자적으로 한 객체에 대한 포인터를 만들고 해당 객체의 reference count를 늘리고, 객체에 대한 포인터를 원자적으로 파괴하고 reference count를 줄일 수 있기를 원합니다. 이런 식으로, reference count가 0이 될 때만 객체를 해제함으로써, 객체가 조기에 해제되지는 않지만, 객체에 대한 포인터가 남아 있지 않을 때 결국 해제되도록 할 수 있습니다.

 위에서 설명한 접근 방식에서 발생하는 주요 어려움은 두 개의 개별 메모리 위치를 원자적으로 수정해야 한다는 것입니다. 포인터와 가리키는 객체의 reference count. 이 원자성을 강화하기 위해 하드웨어 동기화 프리미티브를 사용하거나, 원자가 없는 모양을 달성하기 위해 lock-free 소프트웨어 메커니즘을 사용하여 이를 달성할 수 있습니다. 후자의 접근법에 대한 연구는 다중 변수 동기화 동작의 다양한 lock-free과 wait-free 구현을 산출했다 (예: [20,23]). 그러나 이러한 모든 결과는 복잡하고, 상당한 오버 헤드를 발생시키거나, 정적으로 할당된 메모리의 위치에서만 작동하므로, 동적 크기의 nonblocking 자료구조를 지원하려는 목표에는 적합하지 않습니다. 따라서 이 논문에서 우리는 전자의 접근법을 탐구합니다. 특히, 독립적으로 선택된 두 개의 메모리 위치에 원자적으로 액세스 할 수 있는 DCAS (double compare-and-swap) 명령어의 가용성을 가정합니다. (DCAS는 2 절에서 정확하게 정의되어 있다.) 이러한 명령어는 널리 이용 가능하지는 않지만, 과거에는 하드웨어로 구현되었다(예: [21]). 또한, 확장 가능한 nonblocking 동기화의 실행 가능성을 결정하는 데 하드웨어 동기화 메커니즘의 역할을 조사하는 것이 중요합니다. 따라서 대체 하드웨어 메커니즘으로 이룰 수 있는 것을 연구하는 것이 적절합니다.

 DCAS조차도 항상 정확한 reference count를 유지할 수는 없습니다. 예를 들어, 한 공유 메모리의 포인터가 객체 v를 가리키고, 이 포인터를 다른 객체 w를 가리키도록 변경하면, 포인터를 원자 적으로 수정하고, w의 참조 카운트를 증가시키고, v의 참조 카운트를 감소시켜야 합니다. 그러나, reference count에 대한 약한 요구사항이 충분하며, 이 요구사항은 DCAS를 사용하여 달성할 수 있습니다. 이 약화는 reference count가 항상 정확할 필요는 없다는 관찰에 근거합니다. 중요한 요구사항은 객체에 대한 포인터 수가 0이 아니면, reference count도 같고, 포인터 수가 0이면, reference count가 0이 되어야 한다는 것입니다. 이 두 요구 사항은 각각 객체가 결코 조기에 해제되지 않으며, 포인터가 없는 객체의 reference count가 결국 0이되어 해제될 수 있도록 합니다. [정확하게 말하면, 스레드가 영구적으로 실패하는 경우 garbage가 존재할 수 있고 절대로 해제되지 않을 수 있음을 지적해야 합니다.]

 이러한 관찰은 스레드가 결국 포인터를 생성하거나, 이전의 증가를 보정하기 위해 reference count를 감소시키는 경우, 스레드가 객체에 대한 새로운 포인터를 만들기 전에 객체의 reference count를 증가시키는 것이 안전하다는 것을 의미합니다. 이것은 우리의 방법론에 의해 사용되는 접근법입니다.

reference count에 대해 더 약한 특성이 충분하다는 관찰은 reference count를 유지하기 위해 광범위하게 사용 가능한 단일위치 CAS (single-location Compare-and-Swap) 명령을 사용하는 것을 고려하기에 충분합니다. 그러나, 공유 메모리 위치에서 포인터를 로드 할 때, 로드 된 값이 가리키는 객체의 reference count를 증가시켜야 합니다. 단일위치 CAS로만이 reference count에 액세스 할 수 있는 경우, reference count를 증가시키기 전에 객체가 해제될 수 있으며, 이후에 reference count를 증가시키려고 시도하면 해제되고, 다른 목적으로 재할당된 가능성이 있는 메모리가 손상될 위험이 있습니다. DCAS를 사용하면, 이 개체에 대한 포인터가 여전히 존재하는지 원자적으로 보장하면서 reference count를 늘릴 수 있습니다. 이 시나리오는 reference count를 유지하기 위해 CAS를 사용하는 순진한 접근 방식이 동작하지 않는 이유를 설명하지만, 결코 공식적인 불가능성 주장은 아닙니다. 실제로, 이러한 주장을 구성하는 우리의 후속 작업은 nonblocking 동적 크기 공유 자료구조에서 메모리 관리를 처리하기위한 효과적인 CAS 기반 접근 방식을 만들었습니다 [8-10]. 이 개발은 널리 이용 가능하지 않은 동기화 프리미티브에 기초한 연구가 적절하고 중요하다는 우리의 주장에 더 많은 가중치를 부여합니다.