자바기반 내장형 시스템에서 쓰레기 객체의 명시적 자유화 방법

배 수 강† · 이 승 룡††

요 약

내장형 시스템 소프트웨어의 규모가 커지고 복잡해짐에 따라 동적 메모리 사용이 많아지고, 자동화된 동적 메모리 관리를 수행할 수 있는 쓰레기 수집기의 사용이 보편화 되어가고 있다. 그러나, 쓰레기 수집기의 실행 시 오버헤드로 인하여 발생되는 시스템의 성능저하 문제는 피할 수 없게된다. 본 논문에서는 쓰레기 수집기 사용하는 자바기반의 내장형 시스템에서 실행시간에 쓰레기 수집기로 인한 오버헤드를 줄이기 위한 방안으로 프로그래머가 명시적으로 동적 메모리를 자유화할 수 있는 기법을 소개한다. 제안된 기법은 최상의 경우 쓰레기 수집기가 한 번도 수행되지 않은 채 어플리케이션의 수행이 가능하므로 기존의 쓰레기 수집기로 인한 오버헤드가 전혀 발생되지 않을 수 있다. 반면, 최악의 경우 어떤 쓰레기 객체가 명시적으로 수거되지 않더라도 그것은 추후 쓰레기 수집기에 의해 수거될 수 있기 때문에 쓰레기 수집기를 사용하는 경우와 동일한 오버헤드를 가진다. 제안된 기법은 기존의 모든 쓰레기 수집 알고리즘에 사용될 수 있지만 성능평가 결과 마크-수거 알고리즘에 잘 적용됨을 보여 주었다.

An Explicit Free Method for the Garbage Objects in Java-based Embedded System

Sookang Bae† · Sungyoung Lee††

ABSTRACT

As the size of embedded system software increase bigger and bigger, and it's complexity is grower and grower, the usage of dynamic memory management scheme such as garbage collector also has been increased. Using the garbage collector, however, inherently lead us performance degradation. In order to resolve this kind of performance problem in the Java based embedded system, we introduce an explicit dynamic memory free method to the automated dynamic memory management environment, which can be performed by a programmer. In the worst case, the prosed scheme shows the same performance as the case of that only garbage collector is working, since the unclaimed garbage objects will eventually be collected later by the garbage collector. In the best case, our method is free from any runtime overhead because the applications can be implemented without any intervention of the garbage collector. Although the proposed method can be facilitated with all the existing garbage collection algorithms, it shows an outperform in the case of mark-and-sweep algorithm.

키워드 : 동적 메모리 관리(Dynamic memory management), 쓰레기 수집(Garbage collection), 자바가상머신(Java virtual machine), 자바(Java)

1. 서 론

동적 메모리 관리는 크게 명시적 동적 메모리 관리 기법과 자동화된 동적 메모리 관리 기법으로 나눌 수 있다[1, 2]. 명시적 메모리 관리 기법은 C 언어에서 *malloc/free*와 같은 방식의 동적 메모리를 명시적으로 할당받고 자유화하는 형식과, region이라는 일종의 블록과 같은 영역별로 동적 메모리를 명시적으로 할당받고 자유화는 형식으로 구분할 수 있다[3-6]. 이 기법에서는 프로그래머가 명시적으로 동적 메모리를 할당받고 해제한다. 따라서 정확한 메모리 사용에 대한 제어가 가능하다는 장점이 있지만 프로그래머는 소프트웨어 개발 시 많은 시간을 디버깅을 하는데 소비하는 단점이 있다. 반면에 자동화된 동적 메모리 관리 기법은 프로그래머가 필요할 때 언제든지 동적 메모리를 할당받을 수 있으며, 자유화 기능은 자동화된 동적 메모리 관리자가 대신 수행해주는 방식으로 쓰레기 수집기가 여기에 해당된다. 단순한 내장형 시스템의 경우 정적 메모리의 이용만으로도 소프트웨어의 개발이 가능하지만 내장형 시스템의 운용 프로그램이나 어플리케이션의 규모가 증가하고 프로그램 실행 시 메모리 사용의 특성이 바뀜에 따라 동적 메모리의 사용이 불가피하게 되었다.

한편, 내장형 시스템 분야에서도 소프트웨어 생산성을 높이고, 유지보수 비용을 낮출 수 있으며, 시장진입이 빠른

† 준 회 원 : 경희대학교 대학원 전자계산공학과
†† 종신회원 : 경희대학교 전자계산공학과 교수
논문접수 : 2002년 9월 28일, 심사완료 : 2002년 12월 18일

자바를 많이 사용하고 있다. 무엇보다 자바는 플랫폼의 독립성이 보장되고, 호환성이 보장되며, 자동화된 동적 메모리 관리자를 이용하므로 소프트웨어 개발 시간이 단축되기 때문이다[7, 8]. 썬은 내장형 시스템을 위해 CVM, KVM이라는 자바가상머신을 개발하였다. 자바 사양은 객체의 수거가 자동적으로 이루어질 것을 요구하기 때문에 객체의 수거에 대한 정의를 내리지 않고 있다. 따라서 CVM이나 KVM에서도 일반 JVM과 마찬가지로 쓰레기 수집기를 이용하고 있다.

그러나 자바를 비롯하여 쓰레기 수집기를 지원하는 모든 시스템의 제약점은 프로그래머가 필요에 의해 명시적으로 동적 메모리를 수거할 수 없으며, 쓰레기 수집기의 사용을 강요당하고 있다는 사실이다. 쓰레기 수집기는 많은 편리함과 이득을 제공하지만 실행시간시 무시할 수 없는 시스템 오버헤드를 유발한다는 단점을 가지고 있다.

따라서, 본 논문에서는 명시적 동적 메모리 관리의 장점과 자동화된 동적 메모리 관리의 장점을 모두 수용하여 런타임 시 프로그래머가 명시적으로 쓰레기객체를 자유화하여 쓰레기 수집기의 성능 저하를 최소화시킬 수 있는 기법을 제안한다. 제안된 기법에서는 어떤 객체가 명시적으로 수거되지 않더라도 추후 동적 메모리 관리자에 의해 수거될 수 있기 때문에 최악의 경우 즉, 명시적으로 어떠한 동적 메모리를 수거하지 않은 경우 자동화된 동적 메모리 기법만을 사용한 경우와 동일한 오버헤드를 발생시킨다. 한편 최선의 경우에는 쓰레기 수집기가 한 번도 수행되지 않은 채 어플리케이션의 수행이 가능하므로 기존의 쓰레기 수집기로 인한 오버헤드가 전혀 발생되지 않을 수 있다는 장점이 있다.

본 논문의 구성은 다음과 같다. 2장에서는 본 연구와 관련된 관련 연구에 대해 소개하고, 3장에서는 제안하는 명시적인 동적 메모리 관리기법을 소개하며, 4장에서는 성능평가를, 5장에서는 제안한 기법에 대한 토의 내용을 기술하고, 마지막으로 6장에서 결론과 향후 연구를 밝힌다.

2. 관련 연구

아직 학계나 업계에서 본 논문과 밀접한 관련을 가진 명시적 동적 메모리 관리 기반의 자동화된 동적 메모리 관리에 대한 연구가 진행되고 있지 않지만 본 연구와 관련성이 있는 사례로 Boehm의 *C/C++*을 위한 쓰레기 수집기 라이브러리와 실시간 자바에 대한 두 가지를 소개한다.

Hans Boehm은 쓰레기 수집 기법이 적용되지 않은 기존의 *C/C++*를 위하여 일반 *C/C++* 환경에서도 이용할 수 있는 쓰레기 수집기 라이브러리를 개발하였다[9, 10]. Boehm의 라이브러리에서 *GC_malloc*()이나 *GC_malloc_atomic*() 등의 함수 호출을 통해 생성된 동적 메모리는 Mark-and-Sweep 알고리즘으로 구현된 쓰레기 수집기에 의해 수거될 수 있다. 뿐만 아니라 이러한 동적 메모리는 *GC_free*() 함수를 통하여 명시적으로 수거될 수도 있다[11]. 이 것은 자동화된 동적 메모리 관리자에 의해 할당된 메모리를 어플리케이션이 명시적으로 수거할 수 있다는 점에서 본 연구의 아이디어와 유사하지만 명시적으로 *malloc*()과 같은 함수를 호출하여 얻어진 동적 메모리에 대해서는 쓰레기 수집기가 관리를 할 수 없다는 점이 다르다. 즉, 쓰레기 수집기는 자신의 라이브러리를 이용해 생성된 객체만 관리가 가능할 뿐 *malloc*()과 같은 기존의 함수를 이용하여 얻어진 동적 메모리에 대해서는 관리가 불가능하다는 것이다. 또한 수거하고자 하는 객체가 다른 객체들을 참조하고 있는 경우 이와 같이 참조되고 있는 모든 객체들을 함께 수거할 수 있는 기능은 제공하고 있지 못하다.

Boehm의 라이브러리를 기존의 어플리케이션에 활용하기 위해서 *malloc*() 함수는 *GC_malloc*()으로 대체하고 *free*() 함수는 아무 작업도 수행하지 않도록 하면 쓰레기 수집기가 없는 시스템에서도 쓰레기 수집기의 장점을 제공받을 수 있다는 장점이 있다. 그러나 만약 규모가 큰 기존의 어플리케이션에 Boehm의 라이브러리를 활용하면서 쓰레기 수집기의 오버헤드를 줄이고자 적시적소에 동적 메모리를 수거하고자 하는 경우에는 *free*() 함수에 대한 호출을 일률적으로 빈 작업의 함수로 대체하지는 못하고 필요한 곳에 *GC_free*() 함수로 대체하여야 하는 단점이 있다. 또한 반드시 시스템 *malloc*() 함수를 통해 동적 메모리를 할당받아야만 하고, Boehm의 라이브러리가 관리하는 동적 메모리 영역 내의 객체에 참조가 *malloc*()을 통해 할당받은 메모리 영역 내에 존재하는 경우에는 이를 발견하지 못하고 쓰레기로 취급하여 수거한다는 단점 또한 존재한다. 마지막으로 *GC_free*()를 통해 특정 객체를 수거할 수는 있지만 그 객체가 참조하고 있는 다른 객체들까지 모두 수거해주는 함수는 제공하고 있지 않다. 이는 특히 다단계의 참조 관계로 이루어진 객체 참조의 경우에 트리의 정점이 되는 객체를 수거함으로써 그 객체로부터 참조되는 많은 객체들을 한꺼번에 수거하고자 하는 경우에 유용하게 활용될 수 있으며, 제안하는 기법에서는 이를 지원할 수 있다.

자바 환경에서 프로그래머가 명시적으로 동적 메모리 영역을 수거할 수 있는 방법을 제공한다는 점에서 제안하는 방안과 유사한 실시간 자바가상머신이 있다. 실시간 자바가상머신에 대한 표준은 JCP(Java Community Process)에서 RTSJ(Real-Time Specification for Java)라는 제목으로 JSR(Java Specification Requests)-1에서 진행중이며 2001년 첫 번째 정식 버전을 발표하였다[12]. RTSJ에서 실시간 자바가상머신은 자바 힙 메모리를 제공해야 하며 또한 ScopedMemory, HeapMemory, ImmortalMemory, ImmortalPhysicalMemory 등으로 동적 메모리 영역을 특성에 따라 나누고 이러한 영역에서 메모리를 할당해줄 수 있는 메소드를 가지는 클래스를 제공해야 함을 정의하고 있다. 프로그래머

는 이들 중 어떤 메모리가 필요한 경우 해당 클래스에 대한 객체를 생성하고 그 메모리 영역에 실제 필요한 객체를 생성하게 된다. 따라서 어떤 메모리 영역에 포함되어 있는 메모리 영역 객체가 쓰레기 수집기에 의해 수거되면 해당 메모리 영역이 반환되므로 자연스럽게 객체들이 제거된다.

그러나 RTSJ는 일반 자바 힙을 포함하여 객체가 생성될 수 있는 모든 메모리 영역에 대하여 객체들을 개별적으로 선택하여 수거할 수 있는 방안에 대한 정의는 내리지 않고 있다. 따라서 TimeSys사에서 개발한 실시간 자바가상머신의 참조 구현물[13]을 포함하여 향후 등장하게 될 어떠한 실시간 자바가상머신에서도 자바 언어 자체의 사양이 바뀌지 않은 한 프로그래머에 의한 명시적인 선택적 객체 수거는 지원되지 않을 것으로 예상된다.

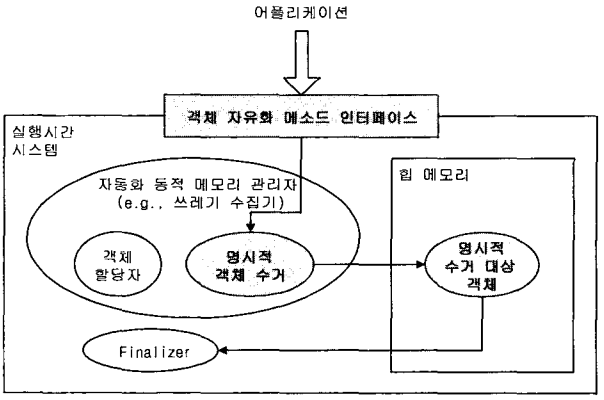
이외에도 실행 시간에 쓰레기 수집기의 오버헤드를 줄이기 위한 방안으로 캐쉬를 고려한 쓰레기 수집 기법도 연구되었으며[14], 실시간 환경에서 쓰레기 수집기의 불명확한 지연 시간 문제를 극복하기 위하여 하드웨어적인 측면에서 쓰레기 수집기를 지원하기 위한 방안에 대한 연구도 진행되었다[15]. 그러나 이들은 쓰레기 수집기가 지원되는 환경에서 프로그래머가 명시적으로 할당받은 동적 메모리 또는 객체를 명시적으로 수거할 수 있도록 하는 본 연구와는 차이가 있다.

3. 자동화된 동적 메모리 관리 기반의 명시적 동적 메모리 관리 기법

이번 장에서는 자바 환경에서 명시적 객체 수거를 위하여 제안된 본 논문의 아이디어에 대한 소개를 한다. 먼저 모델을 도시하고, 본 기법에 대한 제약사항을 나열하며 객체 수거 알고리즘을 설명한다. 또한 자바 수행 환경의 단계별로 객체 수거 과정을 소개한다.

3.1 명시적 객체 수거 기법의 모델

자동화된 동적 메모리 관리 기반의 명시적 동적 메모리 관리를 위한 모델은 (그림 1)과 같다. 언어 자체의 사양을 바꾸지 않고 사용자 어플리케이션이 명시적 객체 수거를 할 수 있도록 본 기법에서는 메소드 형태로 어플리케이션과 자바가상머신간의 인터페이스를 지원한다. 명시적으로 객체 수거 요청이 들어오면 실행시간 시스템 내의 명시적 객체 수거자는 대상 객체가 사용하고 있던 동적 메모리를 해제하고 이를 자유 메모리로 환원한다. 만약 객체가 수거되지 전에 수행해야 될 작업이 있다면 즉, finalize() 메소드를 가지고 있다면 실행시간 시스템의 finalizer가 이 객체를 발견하고 메소드를 호출할 수 있도록 finalizer의 대상에 포함되도록 한다. 따라서 수거 요청된 객체는 쓰레기 수집기의 스캔 및 수거 절차를 거치지 않고도 수거될 수 있기 때문에 수집기의 실행으로 인한 오버헤드를 감소시킬 수 있다.

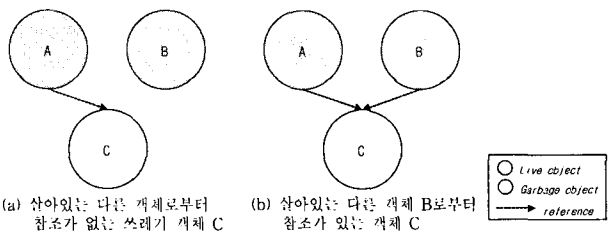


(그림 1) 자동화된 객체 관리 기반 명시적 객체 관리 기법 모델

하나의 어플리케이션에서 많은 객체들은 서로 간의 참조 관계를 유지하며 힙 메모리 내에 존재한다. 따라서 어떤 객체를 수거하고자 하는 경우 그 객체가 다른 객체들과의 참조 관계가 어떻게 되는지를 조사하고 그에 맞는 수거 정책을 정해야 한다. 참조 관계에 따라 다음과 같이 *살아있는 다른 객체로부터 참조의 존재 여부별 수거 정책*과 *수거할 객체가 참조하는 객체에 대한 다른 객체로부터의 참조 여부별 수거 정책*의 두 가지 경우에 대해서 살펴본다.

- **다른 살아있는 객체로부터 참조의 존재 여부별 수거 정책**

어플리케이션에 의해 수거 요청된 객체는 그 객체가 참조되는 상태에 따라 (그림 2)와 같이 두 가지 중 하나의 상태에 있을 수 있다. (a)에서 객체 C는 객체 A로부터의 참조만 존재하며 다른 살아있는 객체들로부터의 참조는 존재하지 않는 경우이다. 따라서 이 경우 객체 A가 객체 C를 수거 요청하는 것은 어플리케이션의 수행에 전혀 영향을 미치지 않기 때문에 문제를 발생시키지 않는다. 객체 A는 객체 C가 확실하게 수거되었다는 사실을 알 수 있지만 (b)의 경우에 객체 B는 객체 C가 수거되었다는 사실을 전혀 모르게 된다. 즉, 또 다른 살아있는 객체로부터의 참조가 있는 객체(C)임에도 불구하고 객체 A는 이를 명시적으로 수거하는 경우이다. 객체 B로부터 참조가 존재하기 때문에 B는 C가 수거된 이후에도 존재하지 않을 객체 C에 향후 메시지를 보내려는 시도를 할 수 있다. 따라서 명시적 수거 관리자는 또 다른 살아있는 객체로부터 참조가 되는 객체를 수거하려는 경우에는 수거 요청을 무시해야 할 것이다.



(a) 살아있는 다른 객체로부터 참조가 없는 쓰레기 객체 C

(b) 살아있는 다른 객체 B로부터 참조가 있는 객체 C

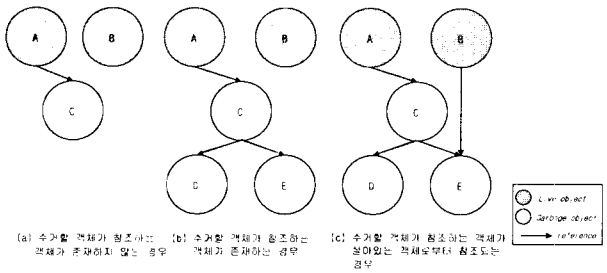
(그림 2) 수거 요청된 객체의 상태

그러나 참조 관계라고 하는 것은 어디까지나 객체간 참조 관계를 물리적으로 보는 것이다. 즉, 살아있는 객체 A가 객체 C에 대한 참조는 가지고 있기는 하나 향후 그 참조를 반드시 이용한다는 보장은 없는 것이다. 따라서 이런 경우에는 수거 요청을 받아들일 수도 있을 것이다. 그러나 향후 참조를 이용하는가 그렇지 않는가를 정확히 판단하는 것은 현실적으로 불가능하다.

그러나 본 논문에서는 프로그래머가 향후 객체간 참조 관계를 이미 확실하게 예상을 하고 명시적으로 객체 수거에 대한 서비스 요청을 하는 것으로 간주하고 이러한 요청을 수락한다. 그런데 만약 프로그래머가 잘못 판단하여 수거된 객체에 메시지를 보내는 경우가 발생한다면 더 이상 그 객체가 존재하지 않음을 예외 객체(자바의 경우, java.lang.Null-PointerException)를 통하여 알리도록 하였다. 따라서 프로그래머는 이러한 예외 정보를 이용하여 잘못된 객체 수거를 올바른 시점으로 수정하여 디버깅에 도움을 받을 수 있게 된다.

- **수거할 객체가 참조하는 객체에 대한 다른 객체로부터의 참조 여부별 수거 정책**

객체 수거시에 (그림 3)과 같이 수거 객체가 참조하고 있는 객체가 있는 경우와 없는 경우의 두 가지를 고려할 수 있다. (a)의 경우 객체 C가 참조하고 있는 객체가 없으므로 단순히 객체 C를 수거하는 것은 문제를 발생하지 않는다. (b)의 경우 수거 요청된 대상이 객체 C인 경우이다. 객체 C는 객체 D, E를 참조하고 있다고 가정하면 객체 C를 수거함으로써 C가 사용하던 메모리는 자유화되어 재사용될 수 있다. 그러나 만약 C가 참조하고 있던 D와 E가 다른 살아있는 객체로부터의 참조가 존재하지 않는다면 이들도 함께 수거할 수 있으며 이 것 역시 문제를 발생하지 않는다.



(그림 3) 수거할 객체가 참조하는 객체에 대한 다른 객체로부터의 참조 여부

제안된 기법에서는 (그림 3)(a) 경우를 위하여 free(reference)와 같은 형태의 인터페이스를 (그림 3)(b)의 경우를 위하여 freeAll(root_reference)의 인터페이스를 제공한다. 그런데 (그림 3)(c)의 경우와 같이 수거 대상인 객체 C가 참조하는 객체 중 어떤 객체(E)가 다른 살아있는 객체(B)로부터 참조가 되고 있는 상태에서 freeAll(C)와 같이 호출하면 문제가 발생할 수 있다. 즉, 객체 E는 B로부터 참조가 되고 있는 객체이기 때문에 참조 관계로만 따지면 살아있는 객체로 보존해야 함에도 불구하고 freeAll()을 이용하게 되면 E는 수거되고 향후 B가 E에게 메시지를 보낼 수 있는 방법이 존재하지 않게 되는 것이다. 이 문제는 3.2에서도 언급되었던 내용과 마찬가지로 프로그래머는 freeAll(C)를 호출할 때 C를 포함하여 C가 참조하는 모든 객체들은 더 이상 프로그램에서 참조되지 않음을 확신하고 호출해야 한다. 만약 프로그래머의 실수로 잘못된 확신 때문에 B가 E의 객체에 메시지를 보내는 경우가 발생된다면 B객체는 java.lang.NullPointerException 예외를 받게 된다. 따라서 프로그래머는 이를 통하여 freeAll(C)를 호출한 것이 잘못된 것임을 확인할 수 있다.

3.2 객체 수거 알고리즘

제안된 명시적 객체 수거 알고리즘은 (그림 4)와 같다. 먼저 라인 1, 2에서는 수거하고자 하는 객체가 살아있는 쓰레드 객체라면 예외 객체(*CannotKillLiveThreadException*)를 만들어 호출한 곳으로 전달한다. 자바에서의 쓰레드는 네이티브 쓰레드에 대한 정보를 저장하고 수행을 제어하는 자바 객체이기 때문에 명시적 수거를 통해 쓰레드 객체는 수거하지 못하도록 하였다. 이와 같은 제약사항을 완화시키기 위한 방안은 향후 연구에서 이루어질 것이다. 그리고 라인 3, 4에서는 수거하고자 하는 객체가 이미 자유화된 객체인 경우 그냥 호출한 곳으로 되돌아간다. 라인 5 이후에서는 살아있는 객체에 해당하는데 그 객체가 *finalize*() 메소드를 가지고 있는 객체라면 라인 6에서 *finalize* 객체 리스트에 추가하고 라인 7에서 모든 어플리케이션 쓰레드의 수행을 중단시키고 라인 8에서 *finalize* 작업을 담당하는 *finalizer* 쓰레드의 수행을 재개한다. 수행을 마치면 다시 어플리케이션의 쓰레드 수행을 라인 9에서 재개한다. 이후 라인 10에서 힙 메모리에 대한 잠금을 얻고, 라인 11에서 이미 구현되어 있는 객체 자유화에 관련된 함수를 호출하여 실제 수거 작업을 수행한 다음, 라인 12에서 힙에 대한 잠금을 풀어준다.

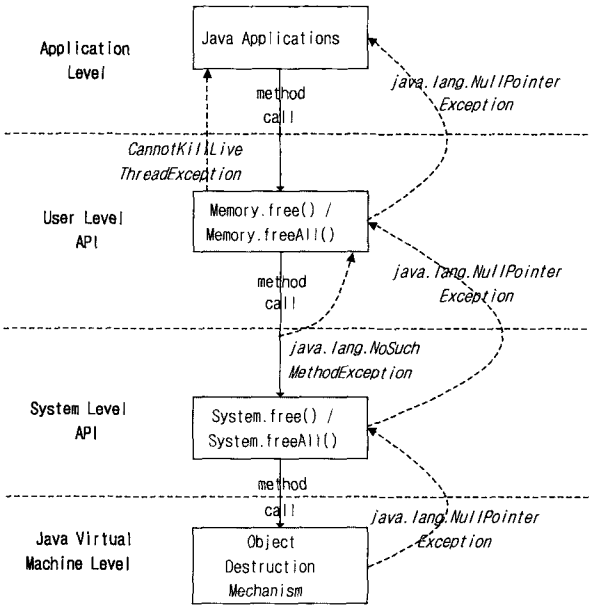
```
1 : If the object is a live thread then
2 :     create a CannotKillLiveThreadException object and
        Throw it
3 : If the object is already freed then
4 :     create a NullPointerException object and Throw it
5 : If the object has a finalize() method then
6 :     add it to the finalize object list
7 :     stop all the application threads
8 :     start the finalizer thread
9 :     resume all the application threads
10 : acquire a lock for the Java heap
11 : call the proper free() function that is already implemented in
     your JVM
12 : release the lock for the Java heap
```

(그림 4) 명시적으로 객체를 제거하는 알고리즘

3.3 자바 실행시간 환경에서 객체 수거의 단계별 구성

명시적인 객체 수거를 지원하기 위하여 제안하는 (그림 1)의 모델을 수거 진행과정의 단계에 따라 (그림 5)와 같이 크게 4개로 구분할 수 있다. 가장 상위 단계에는 명시적으로 객체 수거를 호출하게 되는 자바 어플리케이션이 있고, 가상 머신이 명시적 객체 수거를 지원하면 이를 호출해주고 지원하지 않는다면 아무 작업도 행하지 않는 사용자 레벨의 API 단계가 두 번째 단계에 위치한다. 그리고 세 번째로는 네이티브 메소드로 작성된 시스템 레벨 API가 위치하게 되며, 마지막으로 실제 객체를 수거하는 가상머신 레벨의 객체 수거 메커니즘이 있다.

그림에서 보듯이 자바 어플리케이션에 객체를 수거할 수 있는 서비스를 제공하기 위한 메소드로 *Memory.free*()와 *Memory.freeAll*()가 있다. 전자는 어떤 객체 하나를 수거할 때 호출할 수 있으며, 후자는 인자로 주어진 해당 객체뿐만 아니라 그 객체가 참조하고 있는 모든 객체들도 수거 대상으로 할 때 호출할 수 있다. 앞의 두 메소드는 순수 자바로 구현되어 있으며 상황에 따라 세 가지의 예외 객체를 어플리케이션으로 전달한다. 먼저, 어플리케이션에서 살아있는 쓰레드 객체를 수거하려고 시도하는 경우에는 *CannotKillLive Thread Exception* 예외 객체를 전달하게 되며, 만약 명시적인 객체 수거에 대한 API를 지원하지 않는 가상머신에서 이를 시도한 경우에는 *java.lang.NoSuchMethodException* 예외 객체를 전달하게 된다. *java.lang.NoSuchMethodExpception*은 원시코드의 컴파일 시에 감지될 수 있지만 에러 없이 컴파일된 클래스의 메소드가 명시적인 객체 수거를 지원하지 않는 가상머신에서 관련 메소드를 호출하려는 경우에도 발생될 수 있다. 또한 이미 수거가 된 객체에 대한 접근이 있는 경우에는 *java.lang. NullPointerException*



(그림 5) 객체의 수거에 대한 전체 흐름도

예외 객체를 전달하게 되는데, 이 예외 객체는 앞의 두 메소드에서 직접 생성되어 전달되어지는 것이 아니라 그림과 같이 실제 객체 수거 기능을 담당하는 네이티브 메소드로 구현된 곳에서 생성되어 전달된다. 정상적인 경우 즉, 객체 수거 기능을 할 수 있는 메소드를 가지고 있는 가상머신이라면 그에 대한 적절한 네이티브 메소드를 호출하게 된다.

4. 성능 평가

본 논문에서 제안된 기법은 모든 쓰레기 수집 알고리즘에서 활용될 수 있으나 수집 알고리즘별로 이득이 다를 수 있다. 이번 절에서는 대표적인 쓰레기 수집 알고리즘[16]의 세 가지를 간단히 소개하고 제안된 기법이 적용되었을 때의 이득을 비교한다.

제안된 연구 결과를 검증하기 위해 자체적으로 개발한 유사 자바가상머신에서 제안한 알고리즘을 구현하고 이에 대한 성능 평가를 위해 명시적으로 자유화하고자 하는 각 객체의 개수 및 응용프로그램으로부터 참조를 가지는 객체의 개수에 따라 실험을 하였다. 명시적으로 자유화하고자 하는 객체들은 *finalize*() 메소드를 가지고 있거나 그렇지 않는 객체로 나눌 수 있지만 본 논문에서는 *finalize*() 메소드를 가지고 있지 않은 객체들을 대상으로 하였다. 힙의 크기는 5MB로 고정하였으며, 각각의 경우에 대하여 응용프로그램으로부터 참조를 가지는 객체(L)의 개수를 0개와 30,000개에서 생성되는 전체 객체의 개수를 1백만개부터 천만개까지 증가시켜가면서 응용 프로그램의 수행 시간을 측정하였다. 사용된 하드웨어 시스템은 Pentium-III 600MHz, 256KB 캐시 메모리, 128MB 램의 환경이고, 리눅스의 time을 이용하여 측정하였다. 평가에 사용된 시뮬레이터는 자바가상머신의 기능을 수행하는 프로그램이며, 마크-수거 알고리즘을 위한 버전과 복사형 알고리즘을 위한 버전의 두 가지를 개발하였다.

쓰레기 수집기에 의해서만 쓰레기 객체들이 수거되는 경우를 위하여 (그림 6)의 좌측과 같은 의사 코드 형식의 프로그램을 작성하여 수행하였다. 그리고 프로그래머가 *Memory.free*()라는 메소드를 호출하여 명시적으로 객체를 수거하도록 하는 경우를 위하여 그림의 우측과 같은 의사 코드 형식의 프로그램을 작성하고 이는 명시적인 수거가 지원되는 버전의 시뮬레이터에서 수행하였다. 좌측의 예제 프로그램은 프로그램에서 참조관계를 통해 접근될 수 있는 객체들의 개수를 L에, 참조관계와는 관계없는 쓰레기 객체를 G개만큼 생성한다. 따라서 만약 두 번째 for 루프에서 생성된 쓰레기 객체들의 총 크기가 남은 가용 메모리의 크기를 넘어설 때마다 쓰레기 수집기는 동작을 하게 된다. 우측의 코드는 좌측의 코드와 유사하지만 쓰레기 객체들을 생성하자마자 그 것을 다시 명시적으로 자유화하는 부분이 추가되었다. 따라서 만약 L개만큼의 영구보존용 객체들의 총 크기와 하나의 쓰레기 객체의 크기의 합이 힙 메모리를 넘어서지 않은 경우 쓰레기 수집기는 한 번도 수행되지 않을 수도 있다.

```
L = number_of_persistent_objects
G = number_of_garbage_objects
for i = 1 to L
    create a persistent object
for i = 1 to G
    create a garbage object
```

```
L = number_of_persistent_objects
G = number_of_garbage_objects
for i = 1 to L
    create a persistent object
for i = 1 to G
    create a garbage object
    free the object explicitly
```

(그림 6) 성능 측정을 위한 예제 프로그램의 의사 코드

위 두 가지에 대한 자바 응용프로그램의 전체 수행시간 결과를 쓰레기 수집 알고리즘에 따라 (그림 7)에서 (그림 10)까지 나타내었다. 여기서 L은 매 쓰레기 수집 시마다 쓰레기 수집기가 살아있는 객체로 판단하게 될 객체의 개수이며 (그림 6)의 정수형 L에 해당한다. 또한 G는 (그림 6)의 각각 두 번째 for 문에서 아무런 참조를 가지지 않는 즉, 생성될 쓰레기 객체의 개수가 저장된 변수이다. 쓰레기 수집기는 매 실행주기마다 L개만큼의 객체들을 스캐닝하게 되고 이들은 살아있는 객체로 살아남게 되며, G개만큼의 쓰레기 객체들은 스캐닝에 탐지되지 않고 결국 수거가 될 것이다. 그리고 각 그림에서 범례로 표시한 "GC Free"는 명시적인 객체 수거가 지원되지 않는 일반 자바가상머신에서의 실행한 경우를 의미하며, "Explicit Free"는 명시적 객체 자유화 기법이 지원되는 곳에서의 실행한 경우를 의미한다.

4.1 참조계수 카운트 알고리즘

이 알고리즘은 객체의 참조가 발생될 때마다 그 객체의 참조계수 값을 하나씩 증가시켜나가는 방식으로, 참조계수의 값이 0으로 되면 더 이상 그 객체를 참조하는 객체가 존재하지 않기 때문에 수거하게 된다. 알고리즘이 단순하기 때문에 구현이 용이하다는 장점이 있지만 순환형의 참조를 가지는 객체들의 경우 수거가 불가능하며, 매번 참조 변경 시마다 대상 객체들의 계수 값을 증감시켜야 하는 효율성이 떨어지는 문제 등의 단점이 있다. 따라서 순환형의 참조를 가지는 객체들의 수거가 제대로 이루어지기 위해 복사형이나 마크-수거와 같은 추적형 쓰레기 수집 알고리즘을 추가로 적용하는 변형된 참조계수 카운트 알고리즘이 등장하기도 했으나 본 논문에서는 고려하지 않는다. 순수한 참조계수 카운트 알고리즘에는 명시적 객체 수거 기법이 적용되더라도 시스템에 대한 쓰레기 수집기의 오버헤드는 변함이 없다. 실제적으로 객체의 참조계수의 값이 0으로 되면 곧바로 객체가 수거되기 때문이며, 이 것은 곧 명시적으로 객체를 수거한 경우와 동일하기 때문이다.

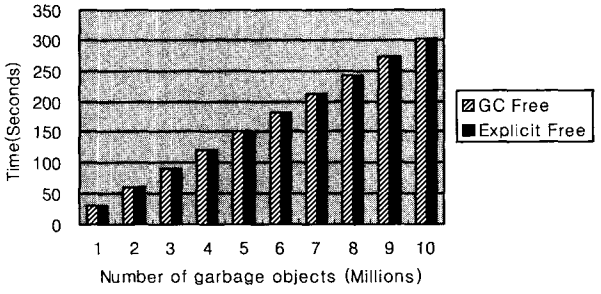
4.2 복사형 알고리즘

복사형 수집기의 경우 전체 동적 메모리를 두 개의 semispace로 나누고 어느 순간에는 단 하나의 semispace 영역만을 활성화시킨다[17]. 이후 할당이 계속 진행되어 현재 활성화된 semipsace가 모두 소비되면 따라서 동적 메모리 이용의 효율성이 50% 이하로 떨어질 수 있다는 단점이 있다. 그러나 마크-수거에 비해 동적 메모리의 할당이 매우 빠르게 정해진 시간 내에 이루어지며, 메모리 단편화 현상이 발생되지 않는다는 장점이 있다.

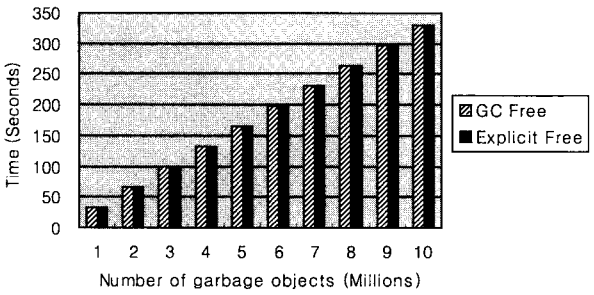
복사형 알고리즘에서 객체의 할당은 현재 활성화된 semispace의 free 포인터가 가리키고 있는 곳에서 이루어지게 되는데 만약 Free 포인터로부터 Top 포인터까지의 크기가 생성요청된 객체의 크기보다 작다면 flip이 일어나 fromspace로부터 tospace로 살아있는 객체들이 복사되며 이들의 역할은 반대가 된다.

복사형 수집기에 명시적 객체 수거 기법을 적용시켜 현재 사용중인 semispace내에 곧바로 사용 가능한 자유 영역을 생성시킬 수 있다. 그러나 Free 포인터를 단순히 증가시키는 복사형 수집기의 할당 정책 때문에 수집기의 한 주기가 실행되기 전에 즉, flip이 일어나기 전에 사용하던 영역을 적절히 재활용할 수 있는 방법이 존재하지 않는다. 결국 이로 인하여 복사형 수집 방식에 명시적 객체 수거 기법이 적용되더라도 별다른 이득을 얻을 수 없을 것이라는 예상을 할 수 있다. 그렇지만 복사형 수집기 원래의 할당 정책을 수정하여 명시적으로 수거된 메모리 영역들의 주소를 큐와 같은 곳에 저장해두고, 더 이상 단순히 Free 포인터를 증가시켜 객체를 생성할 수 없는 상태라면 이러한 큐를 검색하여 가용한 자유 메모리 영역을 찾고 발견되면 재활용할 수도 있다. 그러나 결국 이는 복사형 수집기의 할당이 빠르다는 장점을 약화시키는 요인이 되고 만다.

성능평가를 위하여 (그림 6)의 자바 어플리케이션을 수행하였다. (그림 7)과 (그림 8)의 빗금이 들어간 막대들은 쓰레기 수집기에 의존하여 객체가 수거되는 경우의 수행 시간을, 검은 막대들은 명시적으로 객체를 수거해주는 경우의 수행 시간을 나타낸다. 각 그림에서의 L의 수치는 매번 flip이 발생될 때 시스템 클래스 객체들과 수행하는 어플리케이션 자체의 객체를 제외한 다른 살아있는 객체의 개수를 의미한다. 즉, flip이 발생될 때마다 기존의 semispace에서 다른 semispace로 복사하는 객체의 개수를 의미한다고 볼 수 있다. 그리고 쓰레기 수집기가 수행될 수 있도록 힙의 나머지는 쓰레기 객체들로 채우도록 하였다. 각 그림의 가로축은 이러한 쓰레기 객체의 개수에 따른 시간 결과를 나타낸다. 위에서 예상했던 바와 같이 명시적 객체 수거로 인한 이득이 거의 없음을 알 수 있다.



(그림 7) L=0



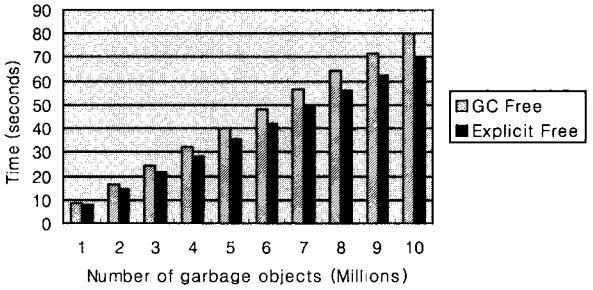
(그림 8) L=30000

4.3 마크-수거 알고리즘

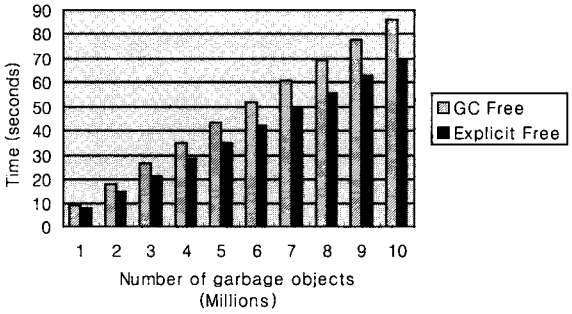
마크-수거(Mark-and-Sweep) 알고리즘은 마킹 단계와 수거 단계의 쓰레기 수집기의 전형적인 두 단계로 이루어지는 단순한 알고리즘이다. 마킹 단계에서는 먼저 루트 집합을 형성하고 이들 객체로부터 참조되어지는 객체들을 마크하는데 또 다시 이들 객체로부터 참조되어지는 객체들을 찾아 마킹하는 단계를 재귀적으로 수행하게 된다. 마킹 단계가 끝나면 마킹이 되지 않은 객체들을 수거하는 수거 단계를 거치게 된다. 이 때 만약 finalize 메소드를 가지고 있는 객체라면 finalizer 쓰레드가 이들을 발견하고 이들 객체들의 finalize 메소드를 호출하고 객체는 생명을 마치게 된다. 순수 마크-수거 알고리즘은 메모리 단편화 현상을 일으키기 때문에 가용한 전체 메모리의 합이 현재 생성하려고 하는 객체의 크기보다 크다고 할지라도 항상 생성 요청된 객체를 할당해줄 수 있는 상태는 아니라는 단점을 가지고 있다. 또한 객체 생성 요청 시에 단편화되어 있는 자유 메모리의 어떠한 영역을 할당해 줄 것인지를 판단하는 알고리즘이 포함되어야 한다.

본 논문에서 소개하는 기법은 참조계수 카운트 기법이나 복사형 수집기의 경우와 달리 마크-수거 알고리즘으로 개발된 쓰레기 수집기와 병행할 경우 가장 큰 이득을 볼 수 있다. 명시적으로 수거되는 객체가 사용하던 메모리는 자유 메모리로 환원되며, 이는 다시 객체 할당자에 의해 곧 바로 재사용이 가능하기 때문이다. 즉, 메모리 단편화 현상을 고려하지 않는다면, 명시적으로 수거되는 객체에 대한 메모리의 크기가 생성 요청된 객체의 크기보다 크기만 하다면 쓰레기 수집기는 결코 수행되지 않게 되기 때문에 오버헤드를 크게 줄일 수 있는 것이다.

성능 평가를 위하여 복사형 수집에서와 같은 상황에서 실험을 하였다. (그림 9)에서 보듯이 순수하게 쓰레기 수집기에 의존하는 경우보다 명시적으로 객체를 수거할 수 있도록 하는 경우가 수행 시간이 더 단축되었다. (그림 9)와 (그림 10)을 비교해보면 한 가지 사실을 추가적으로 알 수 있다. L 즉, 매 쓰레기수집 주기마다 살아있는 것으로 판단되어야 하는 객체의 개수를 증가시킬수록 수집기에만 의존하여 객체를 수거하는 경우에는 어플리케이션의 수행에 필요한 시간은 L에 비례하게 늘어난다는 것이다. 예를 들어 (그림 9)의 가로축 10인 막대를 보면 수행시간이 약 80초 정도이지만, (그림 10)에서의 동일한 위치의 막대를 보면 약 86정도의 수치로 약 6초 정도의 시간 차이를 보이고 있다. 추가 실험으로 L의 개수를 늘릴수록 수행 시간의 차이가 더 커지는 것을 알 수 있었다. 반면 명시적 객체 수거를 이용하고 있는 곳에서는 L의 개수가 늘어나더라도 그 것에 크게 영향을 받지 않음을 알 수 있다. 이는 쓰레기 객체들을 생성하자마자 명시적으로 곧 바로 수거하도록 하여 쓰레기 수집기가 수행될만한 상황을 제공하지 않았기 때문이다.



(그림 9) L=0



(그림 10) L=30000

5. 토 의

5.1 참조의 일관성

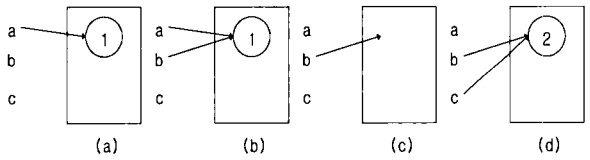
수거하려는 객체에 대한 참조를 또 다른 참조변수가 가지고 있는 경우, 만약 같은 메모리 위치에 같은 타입의 객체가 생성될 때 이는 의도된 참조의 일관성을 깨뜨릴 수 있게 된다. 이에 대한 예제 코드는 (그림 11)에 나타나 있으며, 이것이 실행시 각 참조 변수가 코드의 (그림 11)(a)~(그림 11)(d) 위치에서 자바 힙에 존재하는 객체에 대해 어떤 관계를 가지는지는 (그림 12)에서 보여준다.

위의 자바 소스를 컴파일하고 수행시키면 9, 10 라인과 13, 14 라인에 의해 각각 a=1, b=1과 c=2, b=2라는 결과를 볼 수 있다. *main*() 메소드의 문맥 상 b는 a가 가리키는 객체에 대한 참조를 가지는 변수인데, a에 의해 참조되는 객체를 수거하더라도 b에는 아직 이전의 객체에 대한 주소를 가지고 있다((그림 12)(c)). 공개용 자바가상머신인 Kaffe는 같은 크기의 객체들을 같은 블록에 저장하는 방식을 취하

고 있기 때문에 12라인에서 생성되는 객체는 7라인에서 생성되었다가 11라인에서 수거된 객체의 정확히 같은 위치에 생성된다. 따라서 12라인 이전에서 b를 참조하여 객체에 접근하려는 경우(그림 12)(c)에는 *NullPointerException*이 발생하지만, 12라인 이후에서는 12라인에서 생성된 객체 즉, c가 가리키는 객체를 가리키는 현상이 발생한다(그림 12)(d). 이를 해결하기 위해서는 어떤 객체를 수거할 때 그 객체에 대한 참조를 저장하고 있는 참조변수를 모두 찾아 그들을 null값으로 수정해주어야 한다. 그러나 이렇게 하기 위해서는 전체 힙을 검색하여야 하기 때문에 높은 오버헤드가 발생된다. 따라서 이러한 방식은 현실성이 없기 때문에 다른 대안이 필요하다.

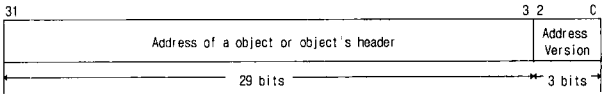
```
1 : public class AnotherReference {
2 :     public int memberVariable ;
3 :     public AnotherReference(int i) {
4 :         memberVariable = i ;
5 :     }
6 :     public static void main(String[] args) {
7 :         AnotherReference a = new AnotherReference(1) ;  // (a)
8 :         AnotherReference b = a ;                         // (b)
9 :         System.out.println("a = " + a) ;
10 :        System.out.println("b = " + b) ;
11 :        System.free(a) ;                                 // (c)
12 :        AnotherReference c = new AnotherReference(2) ;  // (d)
13 :        System.out.println("c = " + c) ;
14 :        System.out.println("b = " + b) ;
15 :    }
16 : }
```

(그림 11) 같은 객체에 대한 다른 참조로 인하여 일관성이 깨지는 예제 코드



(그림 12) 객체 참조의 백업본으로 인한 객체 참조의 불일치 예제

본 연구에서 접근한 일차적인 해결 방법으로는 주소별 버전 정보를 통하여 객체 참조가 원래의 참조와 동일한지 비교하는 것이다. 시스템의 구조에 따라 실제 자바가상머신을 구현할 때 객체의 생성위치를 일반적으로 특정 바이트 단위로 정렬하게 된다. 예를 들어 Kaffe에서는 8바이트 단위로 객체의 시작 주소를 정렬한다. 따라서 객체 참조변수의 하위 3비트는 항상 0값을 가지게 되므로 이 곳에 주소의 버전을 저장하고, 실제 객체가 위치한 곳에도 그 위치의 버전을 저장하여 이들 값을 비교함으로써 참조변수의 주소가 가리키는 것이 실제 참조하던 객체인지 아닌지를 검사할 수 있게 된다(그림 13). 만약 P 위치에 있던 객체 A가 명시적으로 수거되고 다시 같은 위치에 다른 객체 B가 생성된다면 객체 A의 헤더에 있던 주소 버전을 하나 증가시켜 객체 B의 헤더에 저장해둔다. 물론 객체 A를 수거할 때는 헤더에 있던 주소 버전 정보는 삭제하지 않아야 한다. 이후 어떤 객체 B가 자신이 가지고 있던 참조변수를 통해 객체 A에 접근하는 경우, 자신의 참조 변수 값의 하위 3비트 값과 참조 변수가 가리키는 곳의 객체 헤더에 있는 주소 버전을 비교하여, 동일하다면 접근을 허용하고 같지 않다면 (그림 12)(d)에 해당되기 때문에 접근을 허용하지 않도록 한다.



(그림 12) 버전 정보와 실제 객체의 위치로 이루어진 참조 변수 (포인터)

그러나 이는 문제에 대한 정확한 해결 방법은 아니다. 왜냐하면 주소 버전을 저장할 수 있는 공간이 극히 적기 때문에 버전을 정확히 표현하기가 어렵기 때문이다. 어떤 객체에 대한 참조를 가지고 있는 참조 변수가 있는데 이 것이 가리키는 주소의 객체가 반드시 원래 가리키던 객체와 동일하다고 볼 수 없다. 따라서 참조 변수를 일반적인 32비트로 하지 않고 더 넓게 하여 주소 버전으로 이용할 공간을 넓게 하는 것이 해결책이 될 수 있겠으나 이는 메모리 효율성의 또 다른 문제를 발생시킬 수 있다.

5.2 쓰레드에 의해 수행되는 메소드를 가지는 객체의 수거

명시적인 수거를 지원하는 시스템에서는 쓰레드 T_1이 현재 수행하고 있는 메소드를 가지는 객체 O를 다른 쓰레드 T_2가 수거할 경우가 발생될 수 있다. 이를 방지하기 위하여 첫 번째 방안으로는 어떤 객체가 수거 대상으로 지정된 경우에는 모든 자바 스택을 검사하여 각 스택의 최상단에 있는 스택 프레임의 this 포인터가 가리키는 객체와 수거하고자 하는 객체를 비교하여 같은 객체라면 어플리케이션의 이 객체에 대한 수거 요청을 무시하게 할 수 있다. 두 번째 방안으로는 이를 무시하지 않고 어플리케이션이 어떤 객체에 대한 수거 요청이 있었는지를 따로 큐와 같은 곳에 저장해두었다가 모든 쓰레드의 각 스택 프레임이 하나씩 팝업(pop-up)될 때 큐에 저장되어 있는 모든 대상 객체와 검사하여 꺼내어지는 프레임의 this 포인터가 가리키는 객체가 같은 경우라면 그 객체를 명시적으로 수거할 수도 있다. 마지막 세 번째 방안으로는 첫 번째 방안에서의 수거 요청에 대한 무시를 하지 않고 대신 이에 관련된 지정된 예외처리를 하는 것이다.

위와 관련된 경우로써 쓰레드에 의해 향후 수행되어야 하는 메소드를 가지는 객체를 어플리케이션이 수거하려고 하는 경우를 생각해볼 수 있다. 쓰레드가 하나씩 스택 프레임을 꺼내어 가면서 결국 수거된 객체가 this 포인터에 의해 가리켜져있었던 프레임이 최상단에 위치했을 경우 쓰레드는 this 포인터에 의해 가리켜진 객체를 잃어버린 상태이므로 프로그램의 수행이 안전하지 못하게 된다. 따라서 이러한 경우를 위해 수거 요청이 있을 때마다 모든 쓰레드의

각각 자바 스택에 존재하는 모든 스택 프레임의 this 포인터에 의해 가리켜진 객체와 수거 요청된 객체를 비교하여 같다면 이를 무시하게 할 수 있을 것이다. 나머지 방안들은 앞 문단에서의 나머지들과 같다.

5.3 수거된 객체에 대한 잠금을 기다리는 쓰레드

쓰레드 T에 의해 잠금이 걸려있는 객체 O가 있다면 이 객체에 대한 잠금을 얻기 위해 다른 쓰레드들은 블록킹되어 있을 것이다. 쓰레드 T가 동기화되어 있는 메소드 또는 블록을 수행하고 빠져나가면 잠금을 기다리던 쓰레드 중 어떤 하나의 쓰레드가 가상머신의 구현 정책에 따라 선정되어 객체 O에 대한 잠금을 얻고 수행될 것이다. 이 때, 잠금을 얻기 직전에 동기화와 관련되지 않은 어떤 또 다른 쓰레드가 객체 O를 명시적으로 수거하려고 할 수도 있다. 만약 그렇게 되었다면 객체 O에 대한 잠금을 얻기 위해 기다리던 쓰레드들은 영원히 O에 대한 잠금을 얻지 못하고 블록킹될 수도 있다는데 문제가 있다. 이러한 경우에 대한 대책으로는 객체의 잠금을 기다리고 있는 쓰레드가 있는 경우, 그 객체의 수거 요청을 무시하도록 하여 프로그램을 안전하게 수행하거나 또는 지정된 예외 객체를 발생하게 함으로써 프로그래머가 예외를 처리하게 할 수 있다.

6. 결론 및 향후 연구

본 논문에서는 내장형 시스템과 같이 사전에 실행시간의 특성이 비교적 잘 알려진 분야에서 자바를 도입하는 경우 쓰레기 수집기로 인한 오버헤드를 줄이기 위한 방안으로 쓰레기 수집기 기반의 명시적 객체 수거 기법을 살펴보았다. 자동화된 동적 메모리 관리 시스템(특히 쓰레기 수집기)과 명시적인 동적 메모리 관리 시스템에 대한 연구는 각기 따로따로 진행되어 왔다. 그러나 현실적으로는 쓰레기 수집기의 실행시간 오버헤드를 무시할 수 없으며, 또한 명시적인 동적 메모리 관리에 대한 프로그래머의 오버헤드 또한 무시할 수 없다. 따라서 본 연구에서는 이들 두 가지 영역을 적절히 결합함으로써 수집기의 실행시간 오버헤드를 줄일 수 있는 방안을 제시하였다.

안전한 어플리케이션의 수행이 이루어질 수 있도록 참조의 일관성이 없어지는 문제, 쓰레드에 의해 수행되어야 하는 메소드를 가지고 있는 객체를 수거하는 문제, 락을 얻기를 기다리고 있는 쓰레드의 대상 객체를 수거할 때 발생할 수 있는 문제 등에 대한 소개와 이의 해결책 등을 제시하였다. 특히 첫 번째의 경우에는 위치 버전 기법을 이용하여 어느 정도 해결할 수 있으며, 두 번째의 경우에는 자바 스택의 스택 프레임을 조사하여 이를 감지하고 이를 예외 처리하거나 또는 무시 등을 통하여 해결할 수 있음을 보였다. 그리고 세 번째의 경우에는 해제될 수 없는 블록킹 현상을 막기 위하여 객체 수거 요청을 무시하거나 지정된 예외 객체를 발생하게 함으로써 문제를 해결할 수 있게 된다.

향후 연구로는 쓰레기 수집 알고리즘으로 마크-수거를 이용하고 있는 Kaffe 자바가상머신에 제안하는 기법을 실제 적용하고 성능평가를 통하여 본 기법의 우월성을 입증할 것이다. 또한 본 논문에서는 각 쓰레기 수집기 알고리즘에 명시적 기법을 적용하였을 때의 성능을 정성적으로 분석하였지만 향후 Kaffe의 쓰레기 수집 알고리즘을 다른 알고리즘으로 수정하고 각 알고리즘별로 제안하는 기법에 대한 정량적인 분석을 할 것이다. 또한 복사형 수집기의 경우 flip이 발생되기 전 Free 포인터의 위치가 한 방향으로 한 번만 증가하는 형식을 취하고 있기 때문에 명시적으로 수거된 객체의 메모리 영역을 재활용할 수 없다는 단점을 극복하기 위하여 복사형의 할당 방식을 수정하고 이에 대한 성능을 분석할 것이다.

참 고 문 헌

[1] Paul R. Wilson, Mark S. Johnstone, Michael Neely, and David Boles, "Dynamic Storage Allocation : A Survey and Critical Review," International Workshop on Memory Management, Lecture Notes in Computer Science, Vol.986, pp. 1-116, 1995.

[2] Paul R. Wilson, "Uniprocessor Garbage Collection Techniques," In Yves Bekkers and Jaques Cohen, editors, Proceedings of the 1992 International Workshop on Memory Management, pp.1-42, St Malo, France, pp.17 19, 1992.

[3] David Gay and Alex Aiken, "Memory Management with Explicit Regions," In Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation, number 33 : 5 in SIGPLAN Notices, pp.313-323, 1998.

[4] D. T. Ross, "The AED free storage package," Comm. ACM, Vol.10, No.8, pp.481-492, 1967.

[5] Yuuji Ichisugi and Akinori Yonezawa, "Distributed garbage collection using group reference counting," In OOPSLA/ECOOP '90 Workshop on Garbage Collection in Object-Oriented Systems, 1990.

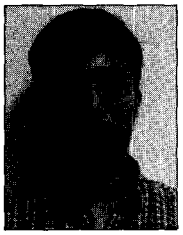
[6] David R. Hanson, "Fast allocation and deallocation of memory based on object lifetimes," Software Practice and Experience, Vol.20, No.1, pp.5-12, 1990.

[7] J. Gosling, B. Joy, and G. Steele, The Java Language Specification, Addison Wesley, Boston, 1996.

[8] T. Lindholm, and F. Yellin, The Java Virtual Machine Specification, Addison Wesley, Boston, 1997.

[9] Hans-Juergen Boehm, and Mark Weiser, "Garbage Collection in an Uncooperative Environment, Software Practice and Experience," Software Practice & Experience, Vol.18, No.9, pp.807 820, 1988.

[10] Hans-Juergen Boehm, "Space Efficient Conservative Garbage Collection," Proceedings of SIGPLAN'93 Conference on Programming Languages Design and Implementation, of ACM SIGPLAN Notices, Vol.28(6), pp.197-206, 1993.
[11] Hans-Juergen Boehm, A garbage collector for C and C++, http://www.hpl.hp.com/personal/Hans_Boehm/gc/index.html.
[12] The Real-Time for Java Expert Group, The Real-Time specification for Java, Addison-Wesley, Boston, 2001.
[13] TimeSys Products and Services - Real-Time Java, http://www.timesys.com/prodserv/java/index.cfm.
[14] Henry G. Baker, "Cache-conscious copying collection," In OOPSLA/ECOOP '91 Workshop on Garbage Collection in Object-Oriented Systems, 1991.
[15] Kelvin D. Nilsen and William J. Schmidt, "A high-performance hardware-assisted real time garbage collection system," Journal of Programming Languages, Vol.2, No.1, 1994.
[16] Richard Jones, and Rafael Lins, Garbage Collection : Algorithms for Automatic Dynamic Memory Management, John Wiley & Sons, England, 1996.
[17] Henry Baker, "List processing in real time on a serial computer," CACM, Vol.21, No.4, pp.280-294, 1978.



배 수 강

e-mail : bsk@oslab.khu.ac.kr
1997년 경희대학교 전자공학과 공학사
1999년 경희대학교 전자계산공학과 석사
1999년~현재 경희대학교 전자계산공학과 박사과정 재학중
관심분야 : 쓰레기 수집기, 자바가상머신, 내장형 시스템, 실시간 시스템

이 승 룡

e-mail : sylee@oslab.khu.ac.kr
1978년 고려대학교 재료공학과 공학사
1986년 Illinois Institute of Technology 전산학과 석사
1991년 Illinois Institute of Technology 전산학과 박사
1992년~1993년 Governors State University, Illinois 조교수
1993년~2001년 경희대학교 전자계산공학과 부교수
2001년~현재 경희대학교 전자계산공학과 교수
관심분야 : 실시간 시스템, 실시간 고장허용시스템, 멀티미디어 시스템