자바 실행시간 환경에서 명시적인 동적 메모리 관리 기법

(An Explicit Dynamic Memory Management Scheme in Java Run-Time Environment)

배 수 강 † 이 승 룡 †† 전 태 웅 †††

(Sookang Bae) (Sung Young Lee) (Taewoong Jeon)

요 약 자바 언어에서 *new*라는 키워드로 생성된 객체들은 C나 C++언어에서의 *free* 또는 *delete*와 같은 키워드를 사용하지 않고 자바가상머신의 쓰레기 수집기에 의하여 자동적으로 관리 (유지 또는 제거)되어진다. 따라서, 응용프로그래머는 메모리 관리에 대한 부담을 전혀 가지지 않고 프로그래밍을 할 수 있다는 장점이 있다. 그러나, 쓰레기 수집기는 자체 실행시간 오버헤드로 인하여 자바가상머신의 성능을 저하시킨다. 이러한 점을 개선하기 위하여, 본 논문에서는 쓰레기 수집기를 사용하는 자바환경에서 프로그래머가 최소한의 프로그래밍 오버헤드를 가지고 명시적으로 객체를 수거함으로 쓰레기 수집기의 실행시간 오버헤드를 줄일 수 있는 방안을 제시한다. 이를 위하여, 제안된 기법에서는 자바 어플리케이션이 순수 자바로 작성된 API를 호출하고, 이것이 다시 가상머신의 종속적인 루틴을 호출함으로써 자바가 가지는 이식성을 그대로 유지하도록 하였다. 다시 말하면 어플리케이션 수행의 안정성은 유지하면서 프로그래머가 단순히 API만을 호출함으로 자바가상머신의 성능향상을 이룰 수 있게 하였다. 마크-수거(Mark-and-Sweep) 알고리즘에 제안한 방법을 적용한 결과 쓰레기 수집기만으로 작동되는 경우의 객체수거 시간에 비해 최저 10%에서 최고 52% 이상의 수행시간 향상을 보였다.

키워드 : 동적 메모리 관리, 쓰레기 수집, 자바, 자바가상머신

Abstract The objects generated by the keyword *new* in Java are automatically managed by the garbage collector inside Java Virtual Machine (JVM) not like using the keywords *free* or *delete* in C or C++ programming environments. This provides a means of freedom of memory management burden to the application programmers. The garbage collector however, inherently has its own run time execution overhead. Thus it causes the performance degradation of JVM significantly. In order to mitigate the execution burden of a garbage collector, we propose a novel way of dynamic memory management scheme in Java environment. In the proposed method, the application programmers can explicitly manage the objects in a simple way, which in consequence the run-time overhead can be reduced while the garbage collector is under processing. In order to accomplish this, Java application firstly calls the APIs that are implemented by native Java, and then calls the subroutines depending on the JVM, which in turn support to keep the portability characteristic Java has. In this way, we can not only sustain the stability in execution environments, but also improve performance of garbage collector by simply calling the APIs. Our simulation study show that the proposed scheme improves the execution time of the garbage collector from 10.07 percent to 52.24 percent working on Mark-and-Sweep algorithm.

Key words : Dynamic memory management, Garbage collection, Java virtual machine, Java

† 비 회 원 : 경희대학교 컴퓨터공학과
bsk@oslab.kyunghee.ac.kr
†† 종신회원 : 경희대학교 컴퓨터공학과 교수
sylee@oslab.kyunghee.ac.kr
††† 종신회원 : 고려대학교 컴퓨터정보학과 교수
jeon@tiger.korea.ac.kr
논문접수 : 2002년 5월 28일
심사완료 : 2002년 10월 14일

1. 서 론

일반적으로 메모리는 컴파일 시에 특성이 결정되는 **정적 메모리**와 실행 시간에 특성이 결정되는 **동적 메모리**로 나눌 수 있다. 이 중 후자의 메모리 관리는 어플리케이션 내에서 명시적으로 동적 메모리를 생성 및 환원하

는 방식인 **명시적 동적 메모리 관리**와 쓰레기 수집기 등의 메모리 관리자를 이용하는 **자동화된 동적 메모리 관리**로 나눌 수 있다[1, 2].

명시적 동적 메모리 관리는 메모리 사용의 효율성 면에서 동적 메모리 관리보다 상대적으로 우월하다는 장점이 있는 반면 프로그래머가 동적 메모리 관리에 대한 모든 책임을 져야 한다는 단점이 있다. 이로 인하여 프로그래머는 소프트웨어 개발 중 많은 시간을 메모리 관련 디버깅에 소비해야하기 때문에 소프트웨어의 생산성이 떨어지게 된다. 반면 자동화된 동적 메모리 관리 환경에서는 쓰레기 수집기의 도입으로 인하여 프로그래머는 메모리 관리에 대한 부담을 전혀 가지지 않게 되므로 소프트웨어 생산성을 증진시킬 수 있다는 장점이 있다. 그러나, 실행시간 시스템은 어플리케이션이 생성한 동적 메모리 또는 객체들에 대한 수거 여부를 실행 시간에 판단해야 하기 때문에 오버헤드가 발생된다. 그럼에도 불구하고 이러한 단점은 장점이 주는 더 많은 이득으로 인해 충분히 감내할 만한 것으로 받아들여지고 있다.

현재 쓰레기 수집기 분야에서 많은 연구가 활발하게 이루어져 과거에 비해 성능 향상이 이루어지고 있기는 하지만 여전히 메모리 관리를 자동화하는데 따르는 오버헤드가 발생되고 있다. 이는 쓰레기 수집기가 실행시간에 쓰레기 객체와 살아있는 객체를 구분해야 하기 때문에 발생되는 것이며, 이러한 오버헤드가 사용자와 시스템간에 빈번한 상호작용이 발생하는 환경에서는 쓰레기 수집기 사용의 가장 큰 저해 요인이 되고 있다.

따라서 본 논문에서는 동적 메모리 또는 객체생명의 유지 여부를 전적으로 쓰레기 수집기에 의존하는 방식을 벗어나 수집기의 역할을 프로그래머가 일부 담당하도록 함으로써 소프트웨어 생산성은 떨어뜨릴 수 있지만 쓰레기 수집기의 실행시간 오버헤드를 줄일 수 있는 방안을 제시한다. 제안된 기법에서는 자바 어플리케이션이 순수 자바로 작성된 API를 호출하고, 이것이 다시 가상머신의 종속적인 루틴을 호출함으로써 자바가 가지는 이식성을 그대로 유지하도록 하였다. 다시 말하면 어플리케이션 수행의 안정성은 유지하면서 프로그래머가 단순히 API만을 호출함으로 자바가상머신의 성능향상을 이룰 수 있게 하였다. 이 때 API는 명시적 동적 메모리 관리에서와 같이 단순화시켜 프로그래머의 불편을 최소화하였다. 이를 실행시간 시스템에서 지원하기 위한 두 가지 방안으로는 명시적 동적 메모리 관리 기반의 자동화된 동적 메모리 관리의 경우와 자동화된 동적 메모리 관리 기반의 명시적인 동적 메모리 관리로 분류할 수 있다. 논문에서는 이 중에 후자를 채택하였는데, 그 이유는 할당받은 동적 메모리가 어플리케이션에 의해 명시적으로 수거되지 않더라도 결국 나중에는 쓰레기 수집기에 의해 수거될 수 있도록 하는 것이 안전하기 때문이다.

제안된 기법의 성능 측정을 위해서 자바 실행시간 시스템을 채택하였다[3, 4]. 자바가상머신의 사양에서는 프로그래머에 의해 명시적으로 생성된 객체의 수거 방식을 자동화해야 한다는 규정이 있으며[4], 일반적으로 이는 쓰레기 수집기를 통하여 구현되기 때문이다. 본 논문에서는 소스코드가 공개되어 있는 Kaffe 자바가상머신을 이용하여 제안하는 아이디어를 구현하였다[5]. 테스트 어플리케이션 수행 시간의 측정 결과 쓰레기 수집 시 살아있는 객체의 수에 따라 약 10%에서 52%의 시간단축 결과를 얻을 수 있었다. 본 논문에서 제시한 기법은 내장형 환경으로부터 엔터프라이즈 환경에 이르기까지 다양한 응용 도메인에서 활용할 수 있을 것으로 기대된다. 먼저 내장형 시스템의 경우 메모리 자원의 제약으로 인하여 본 아이디어를 도입한다면 명시적으로 동적 메모리를 수거하는 만큼의 메모리 효율성을 높이면서 쓰레기 수집의 주기를 길게 하여 수집기로 인한 오버헤드를 줄일 수 있을 것이다. 이러한 특성은 사용자와의 상호 작용성이 중요시되는 개인용 컴퓨터 환경뿐만 아니라 엔터프라이즈 컴퓨팅 환경에서도 쓰레기 수집기를 채택한 시스템의 메모리에 관련된 성능 향상을 이룰 수 있을 것으로 기대한다.

본 논문에서는 위와 같은 하이브리드형 동적 메모리 관리 절차에 대해 소개하고, 이를 진행하는 도중에 그리고 이후에 발생될 수 있는 문제점들과 이들에 대한 해결책을 제시하고자 한다. 논문의 구성은 다음과 같다. 2장에서는 자동화된 동적 메모리 관리 시스템에서 명시적 수거를 지원하는 관련 연구들에 대하여 소개하며, 3장에서는 자바 실행시간 시스템에서 쓰레기 수집기 작동 시 프로그래머가 명시적으로 객체의 수거를 할 수 있는 방법을 기술한다. 그리고 4장에서는 메모리 관리에 대한 부담을 실행시간 시스템과 프로그래머가 나눔으로써 시스템 성능 향상에 어느 정도 기여할 수 있는지에 대하여 평가하며, 마지막 5장에서는 결론 및 향후 연구 방향에 대하여 논의한다.

2. 관련 연구

동적 메모리 관리 기법은 지난 수십 년 동안 연구가 진행되어 왔는데 이는 크게 명시적 할당 및 수거 기법, 영역 기반(region based)의 할당 및 수거 기법, 자동화

된 동적 메모리 관리 기법의 3 가지로 나눌 수 있다[6].

첫 번째 종류로는 C에서의 *malloc()*과 *free()*과 같은 라이브러리를 사용하여 명시적으로 동적 메모리를 관리하는 것이다. 이러한 기법은 정확한 메모리 관리를 제공함으로써 메모리 사용의 효율성 면에서 최대화를 이룰 수 있지만 프로그래머가 필요시 명시적으로 동적 메모리를 할당받고 자유화하는 것을 책임져야만 한다. 즉, 적절한 시기에 자유화하지 않는다면 메모리 누수가 발생될 수도 있으며, 특히 이는 시스템 소프트웨어의 경우에는 심각한 문제를 야기시킬 수 있다. 그러나 제안하는 기법은 프로그래머가 명시적으로 동적 메모리를 수거할 수 있지만 이를 수거하지 않았더라도 최종적으로 쓰레기 수집기에 의해 수거를 보장받기 때문에 메모리 누수와 같은 현상은 발생하지 않는다.

두 번째로는 영역기반 메모리 할당 및 수거 기법이 있다. 이 기법은 Ross[7]에 의해 zone이라는 영역에 객체를 할당하는 기법으로 시작되었는데 이후로 group[8], arena[9] 등의 영역으로 확장되어 널리 사용되었다. 이는 영역 단위로 동적 메모리를 할당 및 수거하는 방식을 취하며, 첫 번째 기법에 비하여 경우에 따라서는 더 빠른 성능을 보이기도 한다. 그러나 이 기법 역시 동적 메모리의 할당 및 수거는 프로그래머에 의해 명시적으로 이루어져야 하기 때문에 근본적으로 첫 번째 방법이 가지는 문제점을 지니고 있다.

세 번째 기법으로는 자동화된 동적 메모리 관리 기법이 있다. 위의 첫 번째와 두 번째의 경우는 오래 전부터 프로그래머의 동적 메모리 관리에 대한 짐을 덜어주어 소프트웨어의 생산성을 증대시킬 수 있는 자동화된 동적 메모리 관리에 대한 연구가 상당한 진척을 보이고 있다. 이 중 특히 쓰레기 수집기[10]에 대한 연구가 활발히 이루어져 최근의 많은 언어에서 이를 지원하고 있다. 그러나 쓰레기 수집기 사용은 앞서 언급한 바와 같이 구현 시 자체의 오버헤드를 가지고 있다. 따라서 어플리케이션의 응답 시간을 개선시키고자 점진형 쓰레기 수집기, 실시간 쓰레기 수집기 및 동시성 쓰레기 수집기 등의 다양한 수집기에 대한 연구가 진행되고 있다[10]. 그러나 이러한 수집기들은 동작 방식이 복잡하여 구현이 용이하지 않아 아직 널리 사용되지 못하고 있다. 이들의 주요 관심사는 수집기의 실행시간 오버헤드를 감소시키기 위함이 아니라 어플리케이션의 반응 시간을 개선하거나 수집기의 활동 시간을 예측하는데 있다. 수집기 자체의 실행 시간 오버헤드를 감소시키기 위해 그리고 메모리 단편화 현상을 극복하기 위해 다양한 쓰레기 수집 알고리즘들이 발표되었는데 대표적으로 참조계수(reference counting), 마크-수거, 복사형(copying) 및 세대별 쓰레기 수집기(generational garbage collector) 등으로 나눌 수 있다. 이들은 각기 장단점을 가지고 있다. 예를 들어, 마크-수거의 경우에는 메모리 효율성 면에서는 복사형보다는 뛰어나지만 복사형에서는 발생되지 않는 메모리 단편화가 발생된다는 문제점을 안고 있다. 물론 이러한 메모리 단편화 현상을 극복할 수 있는 마크-밀집(mark-compact)형 알고리즘이 개발되기도 했지만 마크 및 수거 단계를 거친 후 추가적으로 밀집 단계를 거치기 때문에 추가적인 실행 시간 오버헤드를 유발하여 어플리케이션의 반응 시간이 더욱 느려진다는 단점이 발생된다. 본 연구의 아이디어는 이와 같은 쓰레기 수집 알고리즘을 새롭게 개발하는 것이 아니라 기존의 쓰레기 수집기가 제공되고 있는 시스템에 추가적으로 프로그래머가 명시적으로 동적 메모리를 관리할 수 있는 방안을 제공함으로써 양측의 장점을 동시에 수용하도록 하는 것이다.

한편 Hans Boehm은 쓰레기 수집 기법이 적용되지 않는 기존의 명시적 동적 메모리 관리 방식을 제공하는 C/C++를 위하여 동적 메모리 관리를 자동화해줄 수 있는 쓰레기 수집 라이브러리를 개발하였다[11, 12]. Boehm의 쓰레기 수집 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[13]. 이 것은 자동화된 동적 메모리 관리자에 의해 할당된 메모리를 어플리케이션이 명시적으로 수거할 수 있다는 점에서 본 논문에서 제안된 아이디어와 유사하지만 명시적으로 *malloc()*과 같은 함수를 호출하여 얻어진 동적 메모리에 대해서는 쓰레기 수집기가 관리할 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 불가능하다. 또한 수거하고자 하는 객체가 다른 객체들을 참조하고 있는 경우 이들과 같이 참조되고 있는 모든 객체들을 함께 수거할 수 있는 기능도 없다. 한편, Boehm의 라이브러리를 기존의 어플리케이션에 활용하기 위해서 *malloc()* 함수는 *GC_malloc()*으로 대체하면 쓰레기 수집기가 없는 시스템에서도 쓰레기 수집기의 장점을 제공받을 수 있다. 그러나 Boehm의 라이브러리와 기존의 동적 메모리용 함수를 동시에 사용하는 경우에는 문제점이 발생될 수 있다. Boehm의 라이브러리가 관리하는

동적 메모리 영역 내의 객체에 대하여 오직 *malloc()*을 통해 할당받은 동적 메모리 내에서만 참조가 존재한다면 그 객체에 대한 참조를 발견하지 못하고 쓰레기로 취급하여 수거하는 단점이 있다. 더욱이 *GC_free()*를 통해 특정 객체를 수거할 수는 있지만 그 객체가 참조하고 있는 다른 객체들까지 모두 수거해주는 함수는 제공하고 있지 않다. 이는 다단계의 참조 관계로 이루어진 객체 참조의 경우에 트리의 정점이 되는 객체를 수거함으로써 그 객체로부터 참조되는 많은 객체들을 한꺼번에 수거하고자 할 때 유용하게 활용될 수 있다.

자바 언어 사양에서 객체를 명시적으로 제거할 수 있는 메커니즘에 대한 정의는 아직 내려지지 않았기 때문에 본 논문에서 제안한 기법을 제공하는 자바가상머신은 존재하지 않는다. 즉, *new*를 통해서 객체를 생성할 수는 있지만 이미 생성된 객체를 명시적으로 수거할 수 있는 방법은 없으며, 오직 쓰레기 수집기에 의해서만 생성된 객체가 수거될 수 있다. 다만 자바 환경에서 프로그래머가 명시적으로 동적 메모리 영역을 수거할 수 있는 방법을 제공한다는 점에서 제안하는 기법과 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[14]. RTSJ에서는 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 제공하도록 정의하고 있다. 프로그래머는 이들 중 어떤 메모리가 필요한 경우 해당 클래스에 대한 객체를 생성하고 그 메모리 영역에 실제 필요한 객체를 생성하게 된다. 따라서 하나의 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[15]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[16], 실시간 환경에서 쓰레기 수집기의 불명확한 지연 시간 문제를 극복하기 위하여 하드웨어적으로 쓰레기 수집기를 지원하려는 연구도 진행되었다[17]. 그러나 이들은 쓰레기 수집기가 지원되는 환경에서 프로그래머가 명시적으로 할당받은 동적 메모리 또는 객체를 명시적으로 수거할 수 있도록 하는 본 연구와는 근본적으로 차이가 있다.

3. 하이브리드형 동적 메모리 관리 알고리즘

본 장에서는 소스코드가 공개되어 있는 Kaffe 자바가상머신을 이용하여 자바 어플리케이션에서 명시적으로 객체를 소멸시킬 수 있는 API를 지원하기 위한 방법과 각 API에 대하여 설명한다. 그리고 이 기법을 이용한 경우 얻을 수 있는 이득을 개념적으로 기술한다.

3.1 하이브리드형 동적 메모리 관리 기법의 수거 모델

명시적인 객체 수거를 지원하기 위하여 제안하는 모델은 크게 4개의 단계로 구성된다(그림 1). 가장 상위 단계에는 명시적으로 객체 수거를 호출하는 자바 어플리케이션이 있고, 가상 머신이 명시적 객체 수거를 지원하면 이를 호출해주고, 지원하지 않으면 아무 작업도 행하지 않는 사용자 레벨의 API 계층이 위치한다. 그리고 세 번째로는 네이티브 메소드로 작성된 시스템 레벨 API가 위치하게 되며, 마지막으로 실제 객체를 수거하는 가상머신 레벨의 객체 수거 메커니즘이 있다.

그림 1에서 보듯이 자바 어플리케이션에 객체를 수거할 수 있는 서비스를 제공하기 위한 메소드로 *Memory.*

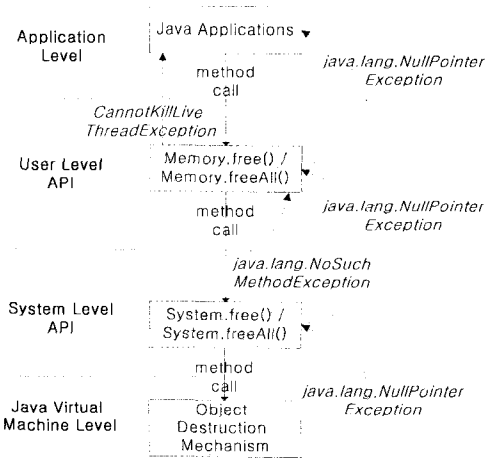

그림 1 객체의 수거에 대한 전체 흐름도

*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.NoSuchMethod Expception*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 수거된 객체에 대한 접근이 있는 경우에는 *java.lang.Null PointerException* 예외 객체를 전달하게 되는데, 이 예외 객체는 앞의 두 메소드에서 직접 생성되어 전달되어지는 것이 아니라 그림 1에서와 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.

그림 2는 그림 1의 객체 수거 메커니즘 부분을 Kaffe에 적용된 예를 보여준다. Java API의 형태로 객체 수거 메소드를 구현하기 위하여 기존의 클래스 중 *System* 클래스를 채택하고 *free()*와 *freeAll()* 메소드를 작성하였으며, 이는 실제 네이티브 메소드로 구현되었다. *System* 클래스와 관련된 함수는 Kaffe의 소스 코드 중 *System.c*에 구현되어 있는데 여기에 *java_lang_System_free()*라는 함수를 정의하고 이 것이 Kaffe의

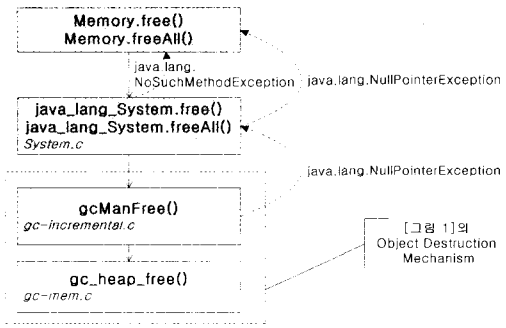

그림 2 그림 1의 객체 수거 메커니즘에 대한 구현 예

쓰레기 수집기 기능이 구현되어 있는 *gc-incremental.c*의 *gcManFree()* 함수를 호출하도록 하였다. Kaffe 가상 머신에서 객체 수준이 아닌 실제 메모리 수준의 수거 기능을 담당하고 있는 함수는 *gc-mem.c*에 있는 *void gc_heap_free(void* mem)*이며, 수거하고자 하는 메모리에 대한 주소를 인자로 취하는데 이는 이미 Kaffe의 개발자들에 의해 구현된 것이다.

3.2 알고리즘 1: 명시적인 객체 수거 알고리즘

제안된 명시적 동적 메모리 관리 알고리즘은 그림 3에서 보여주는데, *gcManFree()*의 구현을 나타낸다. 먼저 수거하고자 하는 객체가 살아있는 쓰레드 객체라면 *CannotKillLiveThreadException* 예외 객체를 만들어 호출한 곳으로 전달한다(라인 1, 2). 그리고 수거하고자 하는 객체가 이미 자유화된 객체인 경우 그냥 호출한 곳으로 되돌아간다(라인 3, 4). 라인 5 이후에서는 살아있는 객체에 해당하는데 그 객체가 *finalize()* 메소드를 가지고 있는 객체라면 *finalize* 객체 리스트에 추가하고(라인 6), 모든 어플리케이션 쓰레드의 수행을 중단시키며(라인 7), *finalize* 작업을 담당하는 *finalizer* 쓰레드의 수행을 재개한다(라인 8). 수행을 마치면 다시 어플리케이션의 쓰레드 수행을 재개한다(라인 9). 이후 힙 메모리에 대한 락을 얻고(라인 10), 이미 구현되어 있는 객체 자유화에 관련된 함수를 호출하여 실제 수거 작업을 수행한 다음(라인 11), 힙에 대한 락을 풀어준다(라인 12).

어떤 시점에 가용 메모리가 부족하여 더 이상의 객체 생성 요구를 수용하지 못할 경우 쓰레기 수집기가 수집 작업을 시작하게 된다. 제안한 기법을 이용하여 명시적으로 불필요한 객체를 자유화하는 경우 크게 두 가지 장점이 있다. 첫째는 메모리의 이득이다. 예를 들어

```
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 3 명시적으로 객체를 제거하는 알고리즘

1M의 힙 메모리가 있으며, 이 중 200KB의 크기에 해당하는 객체들을 명시적으로 자유화하였다고 가정하면 1MB + 200KB의 가용 메모리가 존재하는 것으로 볼 수 있다. 따라서 시스템은 프로그래머가 명시적으로 자유화한 전체 객체들의 크기의 합만큼의 메모리 이득을 얻게 된다. 둘째는 시간의 이득이다. 앞의 예제를 살펴보면 추가로 이용할 수 있는 200KB의 메모리를 어플리케이션의 객체 생성하는데 소비되는 시간만큼의 수집기 시작 시간을 연기할 수 있다. 이로 인하여 예를 들자면, 3번의 수집 작업이 필요했던 어플리케이션이 1번 또는 2번의 수집 작업만으로 수행을 끝마칠 수 있다.

3.3 알고리즘 2: 객체 참조의 일관성을 지원하는 객체 수거 알고리즘

자바가상머신 사양은 힙에서 객체를 어떻게 표현하는가에 대한 정의를 내리지 않고 가상머신 개발자에게 표현 방법을 일임하고 있다. 일반적으로 자바가상머신에서 객체에 접근하는 방법에는 두 가지가 있는데 하나는 핸들을 경유하여 객체에 이르는 방식이고, 다른 하나는 직접 이르는 방식이다[18]. 전자의 예로는 썬 마이크로시스템사의 PersonalJava가 있으며 후자의 예로는 Kaffe가 있다.

핸들을 경유하는 방식에서 자바 프로그램의 객체 참조는 실제 객체를 참조하는 것이 아니라 핸들 풀에 있는 자신의 핸들을 가리키고 있으며, 인스턴스 데이타에 접근하기 위해 다시 핸들에서 가리키는 객체 풀(Pool)을 참조해야 한다. 이 것은 일종의 간접 연결 방식이며 그림 4와 같다. 이 방식의 장점은 복사형 쓰레기 수집기에 의해 인스턴스의 데이타 위치가 옮겨지더라도 핸들의 포인터만 수정하면 되기 때문에 자바 프로그램의 모든 객체참조변수를 검색할 필요가 없다는 점이다. 그러나 객체 접근은 항상 간접 연결 방식으로 이루어지기 때문에 접근 속도가 떨어지는 단점이 있다.

또 다른 방법은 그림 5와 같이 자바 프로그램의 객체 참조변수가 직접 힙의 인스턴스 데이타 영역을 가리키도록 설정하는 것이다. 이 방법의 장점은 객체에 대한 접근이 한 단계로 끝나기 때문에 접근 속도가 빠르다는

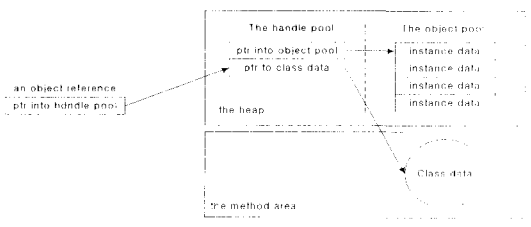

그림 4 핸들 풀과 객체 풀로 나뉘어진 객체 표현

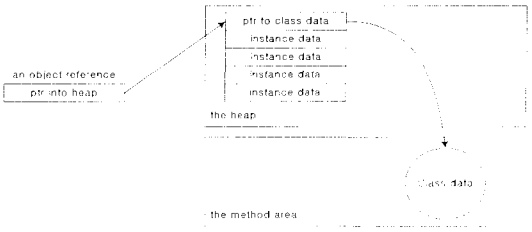

그림 5 하나의 장소에 객체 데이타를 두는 객체 표현

것이다. 그러나 힙의 인스턴스 데이타가 쓰레기 수집기로 인해 다른 장소로 옮겨지게 되면 이를 참조하던 자바 프로그램의 모든 객체참조변수가 가지고 있는 포인터를 수정해주어야 하는 단점이 있다.

본 논문에서 소개하는 기법은 위의 두 가지 방법 중 후자의 방법으로 자바가상머신이 설계된 경우 객체참조의 일관성이 깨어지는 문제를 발생시킬 수 있다. 예를 들어, 자바 프로그램에서 명시적으로 객체를 수거하고 향후 이를 접근하는 경우가 발생되지 않아야 하지만 프로그래머의 실수로 이와 같은 상황이 발생될 수 있다. 따라서 이미 수거된 객체에 대한 접근이 발생하는 경우에 이를 프로그래머에게 알려주어 디버깅에 도움을 줄 수 있도록 명시적 객체수거 알고리즘에 대한 변경이 필요하게 된다.

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 있다. 이에 대한 예제 코드는 그림 6에 나타나 있으며, 실행 시 (a)~(d) 위치에서의 각 참조변수와 자

```
 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 6 같은 객체에 대한 다른 참조로 인하여 일관성이 깨지는 예제 코드

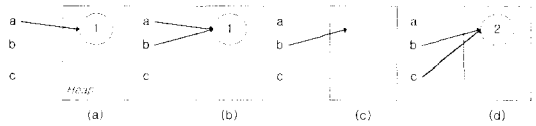

그림 7 객체 참조의 백업본으로 인한 객체 참조의 불일치 예제

바 힙에 존재하는 객체가 서로 어떤 관계를 가지는지는 그림 7에서 보여준다.

위의 자바 소스를 컴파일하고 수행시키면 그림 6의 9,10 라인과 13,14 라인에 의해 각각 a=1, b=1과 c=2, b=2 라는 결과를 얻을 수 있다. *main()* 메소드의 문맥상 b는 a가 가리키는 객체에 대한 참조를 가지는 변수인데, a에 의해 참조되는 객체를 수거하더라도 b에는 아직 이전의 객체에 대한 주소를 가지고 있다(그림 7의 (c)). Kaffe는 같은 크기의 객체들을 같은 블록에 저장하는 방식을 취하고 있기 때문에 12라인에서 생성되는 객체는 7라인에서 생성되었다가 11라인에서 수거된 객체와 정확히 같은 위치에 생성된다. 따라서 12라인 이전에서 b를 이용하여 객체에 접근하려는 경우(그림 7의 (c))에는 *NullPointerException*이 발생하지만, 12라인 이후에서는 12라인에서 생성된 객체 즉, c가 가리키는 객체를 가리키는 현상이 발생한다(그림 7의 (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 많은 오버헤드가 발생된다. 이와 같은 현상은 Boehm의 쓰레기 수집 라이브러리에서도 발생되고 있다.

따라서, 3.2의 알고리즘 1을 이용한 명시적 수거 기법에서 참조의 일관성이 없어지는 상황이 발생되지 않도록 하기 위하여 프로그래머는 수거된 객체를 참조하고 있는 모든 참조에 대한 정리를 직접 수행 또는 관리해야 한다는 부담이 발생된다. 이를 위하여 ***위치 버전*** 기법을 적용하여 객체 참조의 일관성을 없어지는 현상을 완화시킬 수 있는 알고리즘 2를 제안하며, 위치버전이란 다음과 같다.

Kaffe를 포함하여 대부분의 자바가상머신에서는 자바 힙에 존재하는 객체에 접근하기 위하여 포인터를 활용하고 있다. 즉, 32비트 주소 공간에 존재할 수 있는 객체에 대한 접근을 위해 32비트 모두 할당하고 있는 것이다. 그러나 실제 가상머신을 구현할 때에는—예를 들어 Kaffe에서는— 시스템 구조에 따라 메모리 상의 객체 생성위치에 대하여 정렬을 하며, x86계열에서는 8바이트 단위로 객체의 생성 위치를 정렬한다. 따라서 객체 위치가 정렬

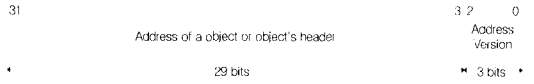

그림 8 버전 정보와 실제 객체의 위치로 이루어진 참조변수(포인터)

되기 때문에 하위 3비트는 항상 0 값을 가지게 되므로 그림 8과 같이 다른 용도로 활용할 수 있으며, 이를 본 논문에서는 ***위치 버전***(또는 ***주소 버전***)으로 정의하고 이는 해당 위치에 생성된 객체의 버전을 저장하게 된다.

위치 버전에는 29비트의 크기로 가리키고 있는 객체가 저장되어 있는 메모리 상의 위치에 대한 버전을 저장하는 용도로 사용된다. 예를 들어 어떤 객체가 가상 머신의 할당 정책에 따라 0x80000000 번지에 생성된다면 그 위치에 저장되어 있는 객체 헤더 정보 중 ***위치 버전*** 정보 값을 가져와 1을 증가시키고 이 것을 다시 객체 헤더에 저장한 뒤, 포인터 값의 하위 3비트에 저장하는 것이다. 따라서 객체가 생성될 수 있는 위치에 대한 버전은 0~7까지 8개의 버전을 지정할 수 있다. 이러한 버전 정보는 나중에 객체가 수거되더라도 그 객체가 위치하던 곳에 버전 정보를 남겨 놓아 추후 이 곳에 새로운 객체가 생성될 때 참조하여 그 값을 1 증가시켜 사용하게 된다.

이와 같은 ***위치 버전***을 이용하면 그림 7의 (d)에서 b와 c는 비록 같은 객체를 가리키더라도 c에 저장된 ***위치 버전***과 실제 객체가 위치한 곳의 ***위치 버전*** 정보는 동일하지만, b에 저장된 ***위치 버전***과 객체의 위치 버전 정보는 다르게 된다. 따라서 가상 머신은 이와 같은 경우를 감지할 수 있게 되고, b라는 참조에 의해 어떤 연산이나 접근이 이루어지는 경우를 제한할 수 있다. 본 논문에서는 이와 같은 상황이 발생될 경우에 *java.lang. NullPointerException* 예외 객체를 전달하도록 구현하였다. 따라서 자바 어플리케이션 개발자는 이러한 예외 객체가 전달될 경우 이미 없어진 객체에 대한 접근 시의 해결책을 "try-catch" 절을 이용하여 구현할 수 있다.

그리고 이와 같은 ***위치 버전*** 방법을 이용하려면 알고리즘 1은 그림 9와 같이 수정되어야 한다. 추가된 라인은 밑줄로 표기하였으며, 객체참조가 가지고 있는 위치 버전 정보와 실제 객체가 가지고 있는 위치 버전이 다른 경우 *NullPointerException*을 생성하여 던지도록 하였다(라인 5,6). 자바 프로그램에서는 빈번한 객체의 참조가 발생되기 때문에 이때마다 가상머신은 29비트의 값을 쉬프트(shift)하여 실제 객체의 위치를 계산하는 방식은 가상머신의 성능을 떨어뜨릴 수 있는 요인이 될 수도 있다. 그러나 쉬프트 연산은 많은 머신 명령 싸이

```
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 position version of the object reference is different from
       the object itself position version then
6:     create a NullPointerException object and Throw it
7: If the object has a finalize() method then
8:     add it to the finalize object list
9:     stop all the application threads
10:    start the finalizer thread
11:    resume all the application threads
12: acquire a lock for the Java heap
13: call the proper free() function that is already implemented in
    your JVM
14: release the lock for the Java heap
```

그림 9 "위치 버전"을 적용한 객체 수거 알고리즘

클을 필요로 하지 않기 때문에 무시될 수 있다.

4. 성능 평가

이번 장에서는 본 연구에서 제안하는 기법이 어떤 쓰레기 수집 알고리즘에서 활용되는 것이 적절한지 살펴보고, 명시적으로 객체를 수거하는 경우와 쓰레기 수집기를 이용한 경우에 대한 실행시간 성능을 비교 분석한다. 이를 위하여 쓰레기 수집의 대표적인 두 가지 알고리즘인 마크-수거 및 복사형 쓰레기 수집 알고리즘에 제안하는 기법을 적용하여 비교 평가한다.

4.1 마크-수거 알고리즘

제안된 연구 결과를 검증하기 위해 자바가상머신에 그림 9의 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 있다. 각각의 경우에 대하여 동일하게 힙의 크기는 5, 10, 20 MB로 고정하였다. 먼저 응용프로그램으로부터 참조를 가지는 객체(L)의 개수를 0개부터 100,000개까지 10,000개 단위로 증가시키고, 각각의 경우에 대하여 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 자바가상머신은 Kaffe 1.0.6 버전이었으며, 디버그 모드로 컴파일 되었다.

쓰레기 수집기에 의해서만 쓰레기 객체들이 수거되는 경우를 위하여 그림 8의 좌측과 같은 코드를 작성하여 명시적인 수거가 지원되지 않는 일반 Kaffe에서 수행하였다. 그리고 프로그래머가 *MM.free()*라는 메소드를 호출하여 명시적으로 객체를 수거하도록 하는 경우를 위하여 그림의 우측과 같은 코드를 작성하고 이를 명시적인 수거가 지원되도록 수정된 Kaffe에서 수행하였다. 좌측의 프로그램을 수행할 때 첫 번째 명령행 인자(*args[0]*)로 주어진 개수만큼 첫 번째 *for* 루프에서 생성된 객체들의 참조가 *gcf*라는 *GCFree*형의 배열 객체에 저장되며 두 번째 *for* 루프에서는 생성된 객체에 대한 참조를 저장하지 않고(이 객체들에 대한 참조를 저장하고 있는 곳이 없으므로 모두 쓰레기 객체들이라고 볼 수 있음) 쓰레기 수집기로 하여금 이들에 대한 수거 작업이 이루어지도록 하고 있다. 우측의 코드가 좌측의 코드와 다른 점은 두 번째 for 루프에서 생성된 객체들(쓰레기 객체들)의 수거를 *MM.free()* 메소드를 이용하여 명시적으로 수거한 것뿐이다.

위 두 가지에 대한 자바 응용프로그램(그림 10)의 전체 수행 시간에 대한 결과는 그림 11부터 그림 14까지 보여준다. 여기서 L은 매 쓰레기 수집 시마다 쓰레기 수집기가 살아있는 객체로 판단하게 될 객체의 개수이며 그림 10의 정수형 L에 해당한다. 또한 G는 그림 10의 각각 두 번째 for 문에서 아무런 참조를 가지지 않는 즉, 생성될 쓰레기 객체의 개수가 저장된 변수이다. 그리고 H는 자바 힙을 의미하며 그 크기를 나타낸다. 쓰레기 수집기는 매 실행주기마다 L개만큼의 객체들을

```
// GCFree.java
public class GCFree {
    int member1;
    public static void main(String[] args) {
        int L = Integer.parseInt(args[0]);
        int G = Integer.parseInt(args[1]);
        GCFree[] gcf = new GCFree[L];
        for (int i = 0; i < L; i++)
            gcf[i] = new GCFree();
        for (int i = 0; i < G; i++)
            new GCFree();
    }
}
```

```
// EXFree.java
public class EXFree {
    int member1;
    public static void main(String[] args) {
        int L = Integer.parseInt(args[0]);
        int G = Integer.parseInt(args[1]);
        EXFree[] exf = new EXFree[L];
        for (int i = 0; i < L; i++)
            exf[i] = new EXFree();
        for (int i = 0; i < G; i++)
            MM.free(new EXFree());
    }
}
```

그림 10 성능 측정을 위한 객체수거 예제프로그램

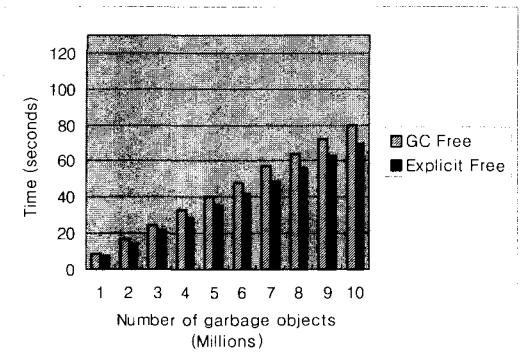

그림 11 L=0(개), H=5(MB)

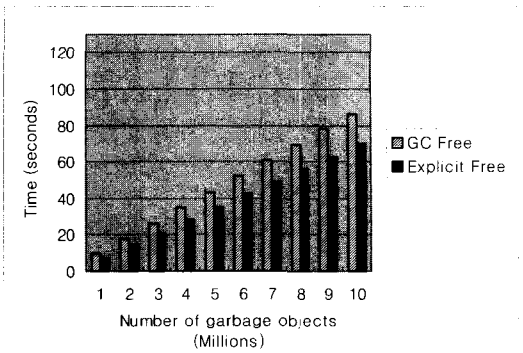

그림 12 L=30,000(개), H=5(MB)

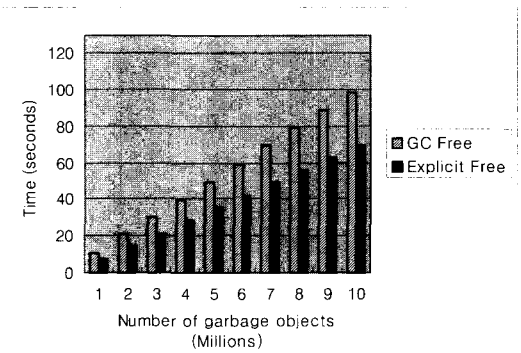

그림 13 L=60,000(개), H=5(MB)

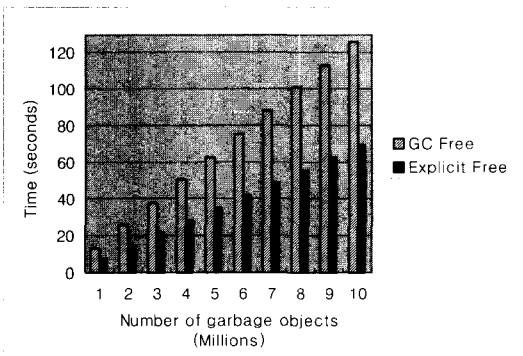

그림 14 L=90,000(개), H=5(MB)

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.

그림 11은 참조를 가지고 있기 때문에 쓰레기 수집기의 매 실행주기마다 살아있는 객체로 판단되어 보존되는 객체(L)의 개수를 0개, 자바 힙을 5MB로 설정한 상태에서 쓰레기 객체(G)의 개수를 1백만에서 1천만까지 증가시키면서 쓰레기 수집기에만 의존한 경우와 명시적 객체 자유화 기법을 적용한 경우에 대한 각각의 수행시간을 측정한 결과이다. 그림에서 보여주듯이 제안된 경우가 더 빠른 수행 속도를 나타내었다.

그림 12는 L의 개수가 3만개로 증가되었다는 것을 제외하고 그림 11의 환경과 동일하다. L의 개수가 증가되었다는 것은 쓰레기 수집기의 매 수행 주기마다 수집기가 작업(스캐닝)해야 하는 양이 증가되었음을 의미한다. 따라서 그림 11에 비해서 쓰레기 수집기에만 의존한 경우 수행 속도가 느려진 것을 쉽게 알 수 있다.

그림 13과 그림 14에서는 L의 개수를 증가시켜 각각 6만, 9만개로 설정하고 성능을 비교 측정하였다. 특히 그림 11과 그림 14를 비교해보면 쓰레기 수집기의 매 수행 주기마다 보존해야 하는 객체의 개수가 증가됨에 따라 쓰레기 수집기에만 의존한 경우는 수집기의 수행 회수가 증가되어 결국 수행 속도가 현저하게 느려지는 것을 알 수 있다. 그러나 명시적인 객체 자유화를 이용한 경우는 살아있는 객체의 개수가 증가되더라도 이 것에 영향을 전혀 받지 않는 것을 볼 수 있다. 명시적으로 자유화한 객체에 의해 사용중이던 메모리의 크기가 어플리케이션이 생성을 요청하는 객체의 크기와 동일하기

때문에 쓰레기 수집기가 한 번도 수행되지 않은 결과이다. 따라서 이는 쓰레기 수집기의 오버헤드를 제외한 어플리케이션의 순수 비용이라고 볼 수 있다. 쓰레기 수집기의 한 주기가 시작되기 전에 어플리케이션에서 명시적으로 객체를 수거하고 그 객체가 차지하고 있던 메모리를 다시 재활용한다면 쓰레기 수집기의 수행을 늦출 수 있으며, 이는 결국 어플리케이션의 전체 수행시간 중 쓰레기 수집기가 수행되는 회수를 줄일 수 있게 된다. 또한 L의 개수가 적은 경우(그림 11)에는 쓰레기 수집기가 동작을 하더라도 살아있는 객체의 개수가 극히 적기 때문에 스캐닝하는 시간이 대단히 짧게 된다. 따라서 명시적으로 객체를 자유화하는 경우와 거의 차이가 나지 않는다는 것을 볼 수 있다. 그러나 이 경우에도 명시적인 경우는 약간의 성능 향상을 이루었다. 이는 비록 쓰레기 수집기가 수행되어 스캐닝하는 객체의 개수가 극히 적지만 수집기 수행의 매 주기다다 수집기 자체의 초기화 및 후처리에 필요한 시간이 필요하기 때문이다.

실험에서는 명시적인 자유화 기법이 쓰레기 수집기에 의한 수거 시간에 비해 약 10.07%에서 52.24%까지 수행 시간 향상을 보였다. 가장 낮은 10.07%의 수행 속도 향상의 경우, 참조를 가지는 객체(L)의 개수는 10,000개이며, 나머지 즉, 아무 의미 없이 생성만 하게되는 객체의 개수는 990,000개인 경우이었다. 그리고 가장 빠른 52.24%의 수행 속도 향상의 경우에 L은 100,000개이고 G는 8,900,000개였다.

그림 15에서 그림 18까지는 그림 11에서 그림 14까지의 경우와 각각 동일하지만 전체적으로 자바 힙의 크기를 10 MB로 설정하고 성능을 측정한 결과이다. 그림 11에서 그림 14까지와 비교해보면 두 가지의 사실을 관찰할 수 있다. 첫 번째는 자바 힙의 크기가 커짐으로써 제안된 기법을 이용한 가상머신과 기존의 가상머신간의 성능 격차가 줄어들었다는 것이다. 이는 쓰레기 수집기

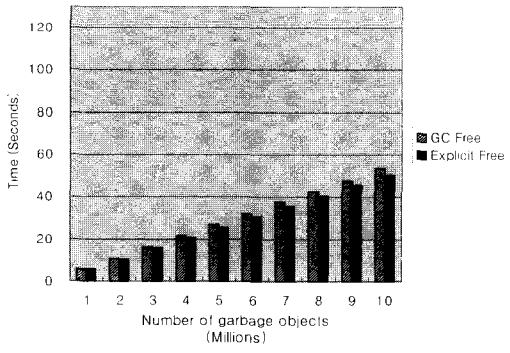

그림 15 L=0(개), H=10(MB)

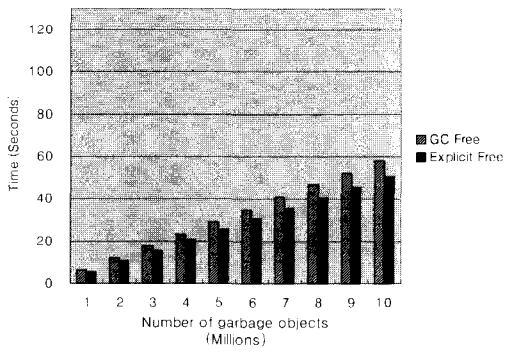

그림 16 L=30,000(개), H=10(MB)

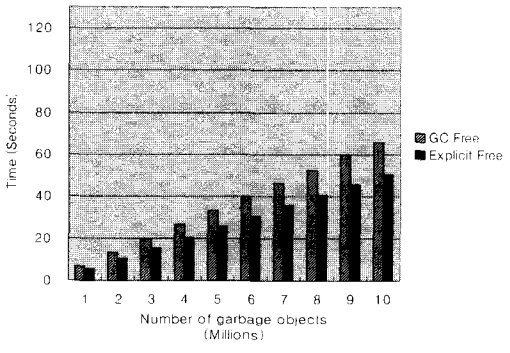

그림 17 L=60,000(개), H=10(MB)

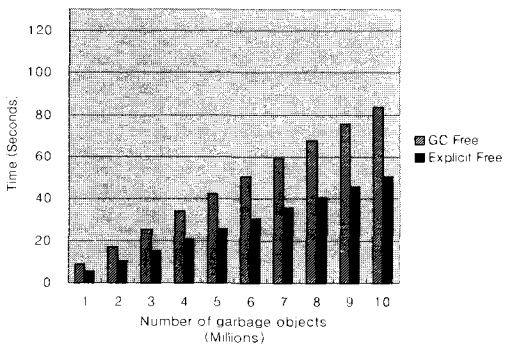

그림 18 L=90,000(개), H=10(MB)

에만 의존한 경우라고 할지라도 자바 힙의 크기가 커졌기 때문에 수집기의 수행 주기가 길어져 결국 수집기의 수행 회수가 줄어들었기 때문이다. 두 번째는 자바 힙의 크기가 커지더라도 제안된 기법을 이용한 가상머신에는 영향을 미치지 않는다는 것을 알 수 있다. 이는 명시적으로 자유화된 객체의 메모리 영역을 곧바로 재활용할 수 있기 때문에 힙의 크기에는 영향을 덜 받는 것이다.

4.2 복사형 수집기

복사형 수집기의 경우 전체 동적 메모리를 두 개의 semispace로 나누고 어느 순간에는 단 하나의 semispace 영역만을 활성화시킨다[19]. 따라서 동적 메모리 이용 효율성이 50% 이하로 떨어질 수 있다는 단점이 있다. 그러나 마크-수거 방식에 비해 동적 메모리의 할당이 매우 빠르게 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.

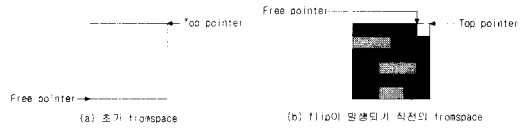

그림 19 복사형 쓰레기 수집기에서 명시적 동적 메모리의 수거 발생 예

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 이 영역([그림 19]-(b)의 음영영역)을 적절히 재활용할 수 있는 방법이 존재하지 않는다. 따라서 명시적 동적 메모리 수거를 하더라도 이로 인한 쓰레기 수집 작업의 지연 효과는 얻을 것으로 예상된다. 이와 같은 가정은 실험을 통해서 증명될 수 있다.

다음 [그림 20]과 [그림 21]은 힙의 크기가 5MB인 동일한 환경에서 각각 L은 0개와 3만개를 가지도록 하였으며, 각각의 경우에 대하여 G는 1백만 개에서 1천만 개까지 증가시키며 어플리케이션의 수행 시간을 측정한 결과이다. 그림에서 나타나듯이 명시적 수거 기법을 이용한 경우와 쓰레기 수집기를 이용한 경우간에 수행시간에 차이가 없다.

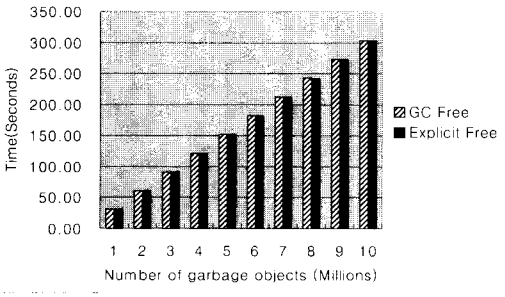

그림 20 L=0(개), H=5(MB)

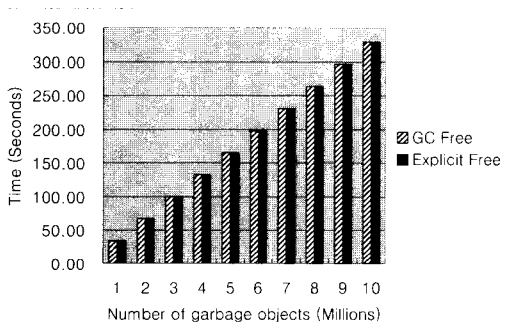

그림 21 L=30,000(개), H=5(MB)

위의 두 가지 경우에 대한 실험 이외에도 힙의 크기를 10MB, 20MB로 증가시키면서 각 경우에 대하여 다양한 크기로 L과 G를 주어 실험을 하였다. 그러나 그 결과는 다음 [그림 22]에서 [그림 25]에서 나타나듯이 명시적 수거 기법을 이용한 경우 및 쓰레기 수집기를 이용한 경우 모두 거의 비슷한 수행 시간을 보여주었다. 다만 [그림 21]과 [그림 22]를 비교해보면 힙 메모리가 증가됨으로써 복사형 수집기의 flip 주기가 길어지게 되어 어플리케이션의 수행 시간이 단축되는 것을 볼 수 있다. 유의할 점은 4.1의 결과와 4.2의 결과를 마크-수거 알고리즘과 복사형 수집의 알고리즘 자체의 성능 평가로 판단해서는 안된다. Kaffe에 현재 구현되어 있는 마크-수거 기법은 다소 최적화가 되어 있으나, 본 실험 결과를 구하기 위해 구현된 복사형 수집기는 최적화되어 있지 않기 때문이다.

복사형 수집기에서 명시적으로 수거된 객체들에 대한 메모리 영역을 쓰레기 수집이 일어나기 전에 재활용하려

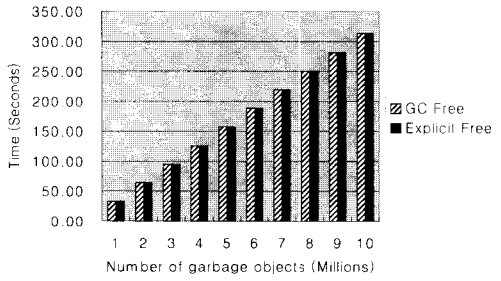

그림 22 L=30,000(개), H=10(MB)

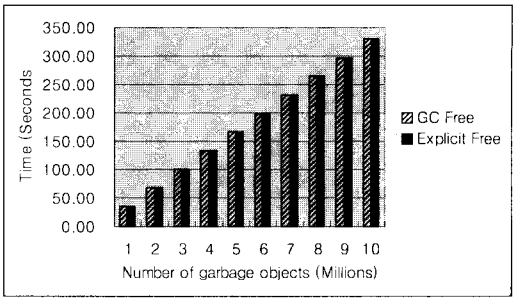

그림 23 L=60,000(개), H=10(MB)

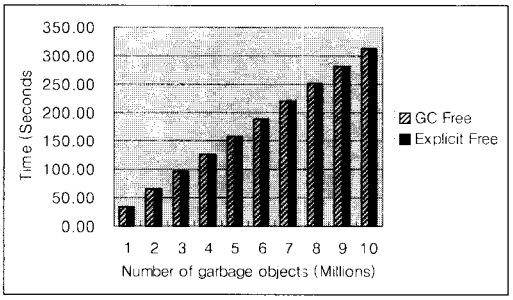

그림 24 L=60,000(개), H=20(MB)

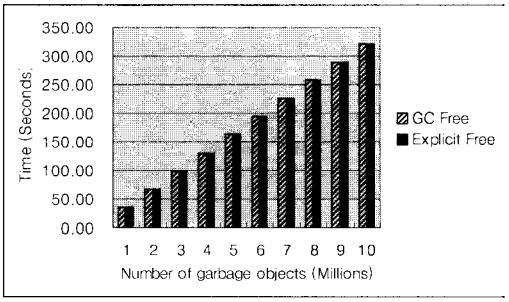

그림 25 L=90,000(개), H=20(MB)

면 근본적인 할당 정책이 바뀌어야 한다. 명시적으로 수거된 메모리 영역들의 주소를 큐나 리스트와 같은 곳에 저장해두고, 더 이상 free 포인터를 증가시켜 객체를 생성할 수 없는 상태라면 이러한 큐나 리스트를 검색하여 가용한 자유 메모리 영역을 찾고 발견되면 재활용할 수도 있다. 검색 방법에는 다양한 메모리 할당 알고리즘이 적용될 수 있으나 이는 복사형 수집기의 할당이 빠르다는 장점을 약화시키는 요인이 되고 만다. 이는 복사형 수집기가 실시간 시스템에서 사용되는 경우, 더 이상 동적 메모리 할당 시간이 정적으로 정해질 수 없음을 말하며 그러한 시스템에서 동적 메모리를 활용하는 태스크들의 수행 시간 예측성이 떨어질 수 있다는 것을 의미한다. 따라서 본 논문에서는 한 번의 쓰레기 수집 주기가 끝나기 전까지는 명시적으로 수거된 메모리 영역이 재활용될 수 없는 복사형 수집기의 근본적인 할당 정책 때문에 이를 이용한 시스템에서의 명시적 객체 수거 지원은 고려하지 않았고, 이는 향후 연구로 남겨둔다.

5. 토 의

5장에서는 본 논문에서 하이브리드형 동적 메모리 관리 기법이 복사형 쓰레기 수집에 활용되는 경우에 대한 고찰과 이러한 기법이 적용된 시스템에서 어플리케이션을 작성할 경우에 자바 프로그래머 또는 가상머신 개발자가 고려해야 할 사항들에 대해 논의한다.

5.1 복사형 수집기에 대한 하이브리드형 동적 메모리 관리의 고찰

복사형 수집기의 경우 전체 동적 메모리를 두 개의 semispace로 나누고 어느 순간에는 단 하나의 semispace 영역만을 활성화시킨다[19]. 따라서 동적 메모리 이용 효율성이 50% 이하로 떨어질 수 있다는 단점이 있다. 그러나 마크-수거 방식에 비해 동적 메모리의 할당이 매우 빠르게 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

이 영역(그림 19-(b)의 음영영역)을 적절히 재활용할 수 있는 방법이 존재하지 않는다. 그러나 복사형 수집기 원래의 할당 정책을 수정하여 명시적으로 수거된 메모리 영역들의 주소를 큐와 같은 곳에 저장해두고, 더 이상 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. 따라서 본 논문에서는 복사형 수집기를 이용한 시스템에서의 명시적 객체 수거 지원은 고려하지 않았다.

5.2 java.lang.Thread 객체의 수거

제안하는 방법을 이용하여 일반 객체가 아닌 쓰레드 객체를 수거하려고 하는 경우가 제한되어야 한다. 만약 현재 수행중인 쓰레드 객체를 수거하는 경우 가상머신의 구현에 따라 어플리케이션이 갑자기 종료될 수 있기 때문이다. 프로그래머의 실수로 쓰레드 객체를 수거하려는 경우이고 그 것이 아직 살아있는 쓰레드인 경우에는 호출한 메소드로 *CannotKillAlive ThreadException*의 예외 객체가 던져지거나 또는 수거 자체를 무시하게 함으로써 프로그램을 안전하게 수행시킬 수 있다.

5.3 쓰레드에 의해 수행되는 메소드를 가지는 객체의 수거

명시적인 수거를 지원하는 시스템에서는 쓰레드 T1이 현재 수행하고 있는 메소드를 가지는 객체 O를 다른 쓰레드 T2가 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 팝(pop)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 예외 처리를 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.

5.4 수거된 객체에 대한 락을 기다리는 쓰레드

쓰레드 T에 의해 락이 걸려있는 객체 O가 있다면 이 객체에 대한 락을 얻기 위해 다른 쓰레드들은 블록킹되어 있을 것이다. 쓰레드 T가 동기화되어 있는 메소드 또는 블록을 수행하고 빠져나가면 락을 기다리던 쓰레드 중 어떤 하나의 쓰레드가 가상머신의 구현 정책에 따라 선정되어 객체 O에 대한 락을 얻고 수행될 것이다. 이 때, 락을 얻기 직전에 동기화와 관련되지 않은 어떤 또 다른 쓰레드가 객체 O를 명시적으로 수거하려고 할 수도 있다. 만약 그렇게 되었다면 객체 O에 대한 락을 얻기 위해 기다리던 쓰레드들은 영원히 O에 대한 락을 얻지 못하고 블록킹될 수도 있다는데 문제가 있다. 이러한 경우에 대한 대책으로는 객체의 락을 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

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 줄이면서 동적 메모리 관리의 자동화를 유지할 수 있는 방안을 제시하였다.

그리고 이 때 안전한 어플리케이션이 되는데 방해될 수 있는 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 번째 문제점은

위치 버전 기법을 이용하여 완화시킬 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 있다.

본 연구에서는 쓰레기 수집기와 명시적 동적 메모리 관리를 결합하는 기법을 소스가 공개되어 있고, 마크-수거 알고리즘으로 쓰레기 수집기가 구현되어 있는 Kaffe 자바가상머신에 적용하였으며, 이를 이용하여 성능평가를 수행하였다. 성능평가를 통하여 최소 10%에서 최고 52%까지 수행 시간 면에서 향상을 보였는데, 특히 수집기가 살아있는 객체로 판단하는 객체의 수가 많으면 많을수록, 그리고 자바 힙의 크기가 적으면 적을수록 큰 성능 향상을 보여주었다. 이는 자바 힙의 크기가 적으면 적을수록 쓰레기 수집기의 수행 주기가 힙의 크기가 큰 경우에 비해 상대적으로 짧아지기 때문이며, 이와 같은 매 수행 주기마다 살아있는 객체가 어떤 것들인가를 판단해야 하기 때문이다.

향후 연구에서는 마크-수거 알고리즘 이외의 다른 알고리즘을 사용하는 쓰레기 수집기와 명시적 동적 메모리 관리 기법에 대한 연구를 진행할 것이다. 예를 들어, 본 논문에서 예로 든 마크-수거 알고리즘에 명시적 동적 메모리 관리 기법을 도입함으로써 쓰레기 수집기의 활동 주기를 늘릴 수 있었고, 이로 인하여 어플리케이션이 전체 수행되는 시간 중 쓰레기 수집기의 수행으로 인한 지연을 줄일 수 있었다. 그러나 복사형 쓰레기 수집기에서는 4.2절에서 살펴본 바와 같이 많은 이득을 얻을 수 없는데, 그 이유는 현재 활동중인 semispace 내에 존재하는 객체들을 명시적으로 수거하더라도 이러한 영역을 재활용할 수 있는 다른 방안(할당 정책)이 강구되지 않는 이상 쓰레기 수집기의 실행 주기에는 변동이 없기 때문이다. 즉, 단순히 현재 free 포인터가 위치한 곳에 새로운 객체를 생성하는 복사형 쓰레기 수집기에서의 할당 정책만으로는 이를 극복하기 어렵다는 것이다. 따라서 하이브리드형 동적 메모리 관리 기법을 위한 복사형 쓰레기 수집기가 될 수 있도록 할당 정책을 수정하는 방안에 대한 연구가 필요하다.

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

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

[5] Kaffe.org, Kaffe Java Virtual Machine, http://www.transvirtual.com and http://www.kaffe.org.

[6] 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.

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

[8] 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.

[9] 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.

[10] Richard Jones, and Rafael Lins, Garbage Collection: Algorithms for Automatic Dynamic Memory Management, John Wiley & Sons, England, 1996.

[11] 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.

[12] Hans-Juergen Boehm, "Space Efficient Conservative Garbage Collection," Proceedings of SIGPLAN '93 Conference on Programming Languages Design and Implementation, Vol. 28(6) of ACM SIGPLAN Notices, pp 197-206, 1993.

[13] Hans-Juergen Boehm, A garbage collector for C and C++, http://www.hpl.hp.com/personal/Hans_Boehm/gc/index.html.

[14] The Real-Time for Java Expert Group, The Real-Time specification for Java, Addison-Wesley, Boston, 2001.

[15] TimeSys Products and Services - Real-Time Java, http://www.timesys.com/prodserv/ java/index.cfm.

[16] Henry G. Baker, "Cache conscious copying

collection," In OOPSLA/ECOOP '91 Workshop on Garbage Collection in Object-Oriented Systems, 1991.
[17] 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.
[18] Bill Venners, Inside the JAVA2 Virtual Machine (2nd edition), McGraw-Hill, 2000.
[19] Henry Baker, "List processing in real time on a serial computer," CACM, Vol.21, No.4, pp.280-294, 1978.

배 수 강
1997년 경희대학교 전자공학과 공학사. 1999년 경희대학교 전자계산공학과 석사. 1999년~현재 경희대학교 전자계산공학과 박사과정 재학중. 관심분야는 쓰레기 수집기, 자바가상머신, 내장형 시스템, 실시간 시스템

이 승 룡
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. 관심 분야는 실시간 시스템, 실시간 고장허용시스템, 멀티미디어 시스템

전 태 웅
1981년 서울대학교 계산통계학과 학사. 1983년 서울대학교 계산통계학과 석사 (소프트웨어공학 전공). 1992년 Illinois Institute of Technology 전산학과 박사. 1992년~1995년 LG산전 연구소 책임연구원. 1995년~현재 고려대학교 전산학과 부교수. 관심 분야는 소프트웨어 테스팅, 객체지향 프레임워크, 소프트웨어 아키텍쳐, 실시간 소프트웨어공학, 실시간 공정제어 시스템