소규모 대화형 시스템을 위한 자바 가비지 콜렉션
(Java Garbage Collection for a Small Interactive System)

권 혜 은 † 김 상 훈 ††
(Hye-Eun, Kwon) (Sang-Hoon, Kim)

요 약 CLDC는 가비지 콜렉션을 위해, 가비지 콜렉션이 필요한 시점에 모든 작업을 수행하는 스탑더월드 가비지 콜렉션 알고리즘을 일반적으로 사용한다. 이 방법은 길고 예측할 수 없는 지연시간으로 인하여 대화형 자바 임베디드 시스템에서는 부적당하다. 본 논문에서는 평균 지연시간을 줄이고 대화형 환경을 지원하는 가비지 콜렉션 알고리즘을 제안한다. 가비지 콜렉터는 객체의 크기에 따라 할당 위치를 결정하는 할당기와 점진적 마크-회수 알고리즘을 사용하는 콜렉터로 구성된다. 가비지 콜렉터는 스레드 스케줄링 정책에 따라 주기적으로 호출되며, 할당기는 콜렉션 주기 동안 마크된 상태의 객체를 할당한다. 또한 콜렉션 주기의 마지막에 비트 패턴의 의미를 교환하는 칼라토글방식을 사용한다. 제안한 가비지 콜렉터와 스탑더월드 마크-회수 가비지 콜렉터의 성능을 비교하였으며, 이 비교 실험을 통해 평균 지연시간은 감소하고, 균일하고 낮은 응답시간을 제공함을 확인하였다.

키워드 : 자바, 프로그래밍 언어, 가비지 콜렉션, 마크-회수 가비지 콜렉션, 점진적 가비지 콜렉션, CLDC

Abstract Garbage collection in the CLDC typically employs a stop-the-world GC algorithm which is performing a complete garbage collection when needed. This technique is unsuitable for the interactive Java embedded system because this can lead to long and unpredictable delays. In this paper, We present a garbage collection algorithm which reduces the average delay time and supports the interactive environment. Our garbage collector is composed of the allocator and the collector. The allocator determines the allocation position of free-list according to object size, and the collector uses an incremental mark-sweep algorithm. The garbage collector is called periodically by the thread scheduling policy and the allocator allocates the objects of marked state during collection cycle. Also, we introduce a color toggle mechanism that changes the meaning of the bit patterns at the end of the collection cycle. We compared the performance of our implementation with stop-the-world mark-sweep GC. The experimental results show that our algorithm reduces the average delay time and that it provides uniformly low response times.

Key words : java, programming language, garbage collection, mark-sweep garbage collection, incremental garbage collection, CLDC

1. 서 론

자바 프로그래밍 언어는 다양한 네트워크 장비와 임베디드 시스템을 위한 프로젝트의 일부에서 비롯되었고, 이후 J2EE, J2SE, J2ME로 분리되었다. 그 중 J2ME (Java 2 Micro Edition)는 소비자 임베디드 장비에서의 자바 프로그램을 위한 것으로 컨피규레이션(Configuration)과 프로파일(Profile)로 구성된다. J2ME의 컨피규레이션 중 하나인 CLDC(Connected, Limited Device Configuration)는 16비트나 32비트 프로세서와 160KB~512KB의 제한된 메모리를 가지면서 네트워크에 연결될 수 있는 장비들을 위해 자바언어와 가상머신 그리고 핵심 라이브러리로 구성되는 최소한의 자바 플랫폼을 정의한다. CLDC를 지원하는 이러한 자바가상머신은 자바가상머신 명세를 따른다.[1, 2, 3]

자바가상머신은 자바 프로그램을 실행하기 위한 여러 가지 실행시간 데이터 영역을 가지는데, 그 중 힙 영역은 다양한 객체들을 할당하기 위해 사용된다. 힙 영역에

· 이 논문은 한국과학재단의 특정기초연구(과제번호 : 1999-1-303-003-3) 지원에 의한 것임.
† 정 회 원 : 세명대학교 대학원 전산정보학과
smallpig@plac.semyung.ac.kr
†† 종신회원 : 세명대학교 컴퓨터정보학부 교수
ksh@plac.semyung.ac.kr
논문접수 : 2002년 7월 8일
심사완료 : 2002년 9월 24일

서 더 이상 사용되지 않는 객체가 차지한 공간은 가비지 콜렉터로 알려진 자동 메모리 관리 시스템에 의해 재생된다.[4] 가비지 콜렉터는 더 이상 메모리 할당이 불가능한 시점에 프로그램의 실행을 중단하고 콜렉션을 실행하는 스탑더월드(stop-the-world)방식과 이 방식의 예측할 수 없는 지연시간 문제를 해결하기 위해 전체 콜렉션 작업을 분리하여 실행 프로그램과 번갈아 실행하는 점진적(incremental) 방식이 있다.[5]

본 논문에서는 CLDC를 지원하는 자바가상머신인 KVM의 가비지 콜렉터를 사용자의 행동을 지원하기 위한 대화형 시스템(interactive system)에 적용 가능하도록 개선하였다. 개선된 가비지 콜렉터는 메모리의 효율적인 사용을 유도하기 위해 객체의 크기에 따라 할당 위치를 결정하는 할당기와 평균 지연시간을 감소하기 위해 전체 작업을 분할하여 실행하는 점진적 마크-회수(mark-sweep) 알고리즘으로 구성된다. 실험을 통해 본 논문에서 제안된 가비지 콜렉터로 인한 지연시간이 기존의 스탑더월드 방식보다 전체적으로 작고, 비교적 일정함을 확인하였다.

본 논문의 구성은 2장 관련연구에서 기존의 가비지 콜렉터에 관해 살펴보고, 3장에서는 점진적 마크-회수 가비지 콜렉터의 구조와 동작 방법에 관해 설명한다. 4장은 동일한 할당기를 사용하는 스탑더월드 방식과 점진적 방식의 콜렉터에 대한 성능 비교, 5장은 결론으로 이루어져 있다.

2. 관련연구

2.1 대화형 시스템과 가비지 콜렉터

사용자와 컴퓨터간 통신을 지원하는 대화형 시스템에서 사용자는 사용자 인터페이스를 통해 어떤 행동을 하고 시스템은 이에 대해 적절히 반응한 후 다음 행동을 기다린다. 이러한 대화형 시스템은 사용자 프로그램이 오랫동안 중단되는 것에 대해 비교적 관대하지만 너무 자주 발생하지는 않아야 한다.[6, 7] 가비지 콜렉터에 의해 사용자 프로그램이 오랫동안 중단되는 것을 해소하기 위한 방법은 가비지 콜렉터의 작업을 작은 조각으로 분리하는 것이다. 작업을 분리하는 첫 번째 방법은 콜렉션의 대상이 되는 힙 영역을 분할하는 경우로 Train 알고리즘이 있다. 이는 메모리를 작은 블록으로 나누어 한번에 한 블록씩 콜렉션하는 방식으로 HotSpot에서 점진적 가비지 콜렉션을 위해 사용된다.[7, 8] 두 번째 방법은 가비지 콜렉터의 작업 자체를 나누어 실행하는 것으로 참조계수(reference-counting) 알고리즘이 여기에 속한다. 참조계수 알고리즘에서 객체는 자신을 참조하는 포인터의 개수를 기록하는 공간을 가진다. 프로그램의 실행에 따라 콜렉터는 참조계수를 확인하고 조정하는 단계와 참조계수가 0인 경우 회수하는 단계로 이루어진다. 이는 실시간 조건을 쉽게 만족할 수 있지만 항상 효과적인 것은 아니며, 구현이 어렵다는 문제점을 가진다.[5, 7]

2.2 점진적 가비지 콜렉터

스탑더월드 방식의 콜렉터는 메모리 할당 요청이 만족되지 않은 경우 사용자 프로그램의 실행을 중단하고 가비지가 점유한 공간을 회수한다. 이에 반해 점진적 방식의 콜렉터는 전체 작업을 여러 개의 조각으로 나누어 하나의 프로세서에서 실행 프로그램과 번갈아 실행한다. 콜렉터와 실행 프로그램이 번갈아 실행됨으로써, 콜렉터가 루트 셋으로부터 도달 가능한 객체를 확인하는 그래프의 운행(trace)을 종료하기 전에 실행 프로그램에 의해 그래프가 변경될 수 있다. 이러한 이유로 실행 프로그램을 뮤테이터(mutator)라고 하며, 콜렉터와 뮤테이터 간 그래프를 일치시키기 위한 방법이 필요하다.

점진적 가비지 콜렉션에서의 객체 상태에 대한 이해를 돕기 위해 Tricolor Marking이 사용되는데, 이는 객체의 상태를 검은 색, 회색, 흰색으로 나타낸다. 검은 색은 콜렉터에 의해 확인되고 마크된 객체이며, 흰색은 아직 발견되지 않은 객체를 의미한다. 회색은 부모노드가 검은 색이고, 자손 노드는 흰색인 객체이다. 콜렉션이 진행됨에 따라 객체는 흰색에서 회색으로 바뀌고 콜렉션의 마지막 시점에 라이브 객체는 검은 색이 되고 흰색 객체가 점유한 메모리는 회수된다.

콜렉터와 뮤테이터 사이의 그래프 일치를 위한 방법에는 읽기 경계(read barrier)와 쓰기 경계(write barrier)가 있다. 읽기 경계는 뮤테이터가 흰색객체를 읽을 때 이 객체를 회색으로 변경하여 뮤테이터가 흰색 객체를 읽을 수 없도록 하는 것이다. 쓰기 경계는 포인터들의 모든 쓰기를 확인하는 것이 요구된다. 읽기 경계와 쓰기 경계는 동일한 잠재적인 복잡성을 가지지만 읽기 보다 쓰기 연산이 더 적다는 사실 때문에 쓰기 경계가 더 효율적이다.[5, 7, 9]

2.3 기존의 점진적 가비지 콜렉션 알고리즘

기존의 콜렉션 방식 중에서 점진적으로 그래프를 운행하여 가비지 콜렉션을 수행하는 알고리즘에는 다음과 같은 것들이 있다. 먼저 Baker의 가장 잘 알려진 Incremental Copying 알고리즘은 Copying 알고리즘에 뮤테이터와의 협력을 위한 읽기 경계를 채택하였다. 콜렉션은 플립에 의해 시작되고 실행 프로그램에 의해 접근되어야 할 객체가 from-space에 존재하는 경우 이를

to-space로 복사한 후, 즉 회색으로 만든 후 접근하는 방식이다. 콜렉션 중 할당되는 객체는 검은 색으로 to-space에 할당된다. Replication Copying 알고리즘은 Baker의 방식과는 다르게 도달 가능한 객체가 to-space로 복사되는 동안 실행 프로그램은 from-space의 객체에 접근한다. 이후 플립이 발생하면 to-space의 객체에 접근하는데, 일관성 유지를 위해 그 동안의 모든 갱신을 발견하여 to-space에 전파할 필요가 있다. 이는 함수 언어를 위해 적합한 방식이다. Yuasa의 Snapshot-at-beginning 알고리즘은 객체에 쓰기 연산이 발생한 경우 이전 포인터 값을 별도로 저장하고 이후에 탐색한다. 이로 인해 콜렉터가 그래프를 운행하는 도중에 그래프가 변경되더라도 도달 불가능한 상태의 객체가 발생하지 않는다. 콜렉션 중 할당되는 객체는 검은 색이다. Incremental Update 알고리즘은 대부분의 객체가 짧은 생존기간(lifetime)을 가진다는 이론을 바탕으로 하여 흰색으로 객체를 할당한다. 할당된 객체가 탐색이 종료된 이후에도 라이브 상태인 경우 회수되지 않도록 하는 처리가 필요하다.[5, 7]

2.4 KVM에서의 가비지 콜렉션

KVM에서 할당 가능한 메모리 영역은 블록의 시작 주소 순서에 따라 링크드 리스트로 연결되는데, 이를 프리 리스트라고 한다. 할당기는 First-fit 정책에 따라 공간을 할당하는데, 이는 단순히 리스트의 시작 부분부터 탐색하여 요청을 만족하기에 충분히 큰 첫 번째 자유 블록을 사용한다. 블록은 요청된 크기보다 크다면 분할되고, 나머지는 프리 리스트에 남는다. 이러한 방식은 프리 리스트의 시작 부분에서 블록의 분할이 많이 발생하여 이후에 큰 블록을 탐색하고자 할 때 탐색 시간이 증가되는 단점을 가진다. 더 이상 할당 가능한 공간이 없는 경우 가비지가 점유한 공간을 회수하여 재사용하기 위해 가비지 콜렉터가 호출된다. 가비지가 차지한 공간을 회수하기 위한 알고리즘은 마크-회수 방식이다. 이는 루트 셋으로부터 도달 가능한 모든 객체를 표시하는 마크 단계와 메모리의 시작부분부터 모든 객체의 마크 여부를 검사하여 마크되지 않은 객체에 의해 점유된 메모리 공간은 회수하고, 그렇지 않은 객체의 마크는 삭제하는 회수 단계로 이루어진다. 이 알고리즘은 메모리 전체 공간의 가비지를 회수하는 동안 실행 프로그램이 중단되는 스탑더월드 방식으로 동작하므로 사용자는 예측 불가능한 시간동안 대기하여야 한다.[5, 12, 13]

2.5 JVM에서의 점진적 가비지 콜렉션

J2EE, J2SE를 위한 HotSpot 가상머신은 메모리 관리를 위해 다양한 기법들을 사용한다. 전체 메모리는 Generational 알고리즘에 의해 young 영역과 old 영역으로 구분되어 관리된다. 객체는 young 영역내의 eden 영역에 할당되는데, 더 이상 객체를 할당할 수 없는 경우 라이브 객체는 Copying 알고리즘에 의해 young 영역의 survivor 영역으로 복사된다. Young 영역보다 비교적 큰 old 영역에 대해서는 선택적으로 점진적 가비지 콜렉션이 실행될 수 있다. 이 때 사용되는 Train 알고리즘은 메모리를 고정된 크기의 car와 이로 구성되는 train으로 구분하여 한 블록씩 콜렉션 한다. 참조되는 객체들은 규칙에 따라 동일 train 내부의 마지막 car 또는 다른 train으로 복사되고 외부로부터 참조되지 않는 car와 train은 회수된다. Train 알고리즘은 경성 실시간 시스템(hard real-time system)을 위한 것이 아니며, 콜렉션으로 인해 사용자 프로그램이 중단되는 시간은 대화형 시스템에 적합하다.[5, 9, 10, 14] 메모리 영역을 분할하고 해당 영역의 라이브 객체를 다른 공간으로 복사하는 이러한 방법들은 가비지 콜렉션이 발생했을 때 라이브 객체를 복사할 공간을 필요로 하고 이를 위해 분할된 공간의 일부만을 사용한다. 그러므로, 메모리 크기에 대한 제약 조건을 가지는 시스템에서 사용하기에는 적합하지 않다.

3. 점진적 마크-회수 가비지 콜렉터

본 논문의 점진적 마크-회수 가비지 콜렉터는 CLDC를 지원하는 자바가상머신인 KVM의 가비지 콜렉터를 소규모 대화형 시스템에 적합하도록 개선한 것이다. 소규모 메모리를 가진 시스템에서 실행된다는 점을 고려하여 메모리의 사용 효율을 높이기 위해 메모리를 분할 대신 콜렉터의 작업을 분할하는 점진적 마크-회수 알고리즘을 사용한다. 그리고, 사용자의 행동을 기다리는 동안에도 필요에 따라 가비지 콜렉션이 발생될 수 있도록 하였다.

3.1 가비지 콜렉터의 실행

가비지 콜렉터의 실행은 스레드의 스케줄링 정책과 연관된다. 환형 리스트로 연결된 스레드들은 라운드 로빈(round-robin) 방식으로 스케줄링 되는데, 가비지 콜렉터는 가장 마지막 스레드의 실행이 완료된 후 실행된다. 이후 가비지 콜렉터의 실행이 종료되면 다시 첫 번째 스레드가 실행된다. 회수 알고리즘은 2.4절에서 설명된 것과 동일한 마크-회수 알고리즘을 사용한다. 그러나 점진적 방식으로의 개선을 위해, 전체 알고리즘은 세부적으로 분할되었다. 자바가상머신의 실행동안 콜렉터가 호출되면, 전체 알고리즘을 분할한 일부만을 수행한다.

3.2 할당 영역과 할당기

마크-회수 기법은 할당된 객체의 위치를 변경하지 않

고, 객체들은 서로 생존기간이 다르므로 할당 가능한 블록은 연속적이지 않을 수 있다. 할당 가능한 인접하지 않은 각 블록은 이중 링크드 리스트로 구성되는데 이를 프리 리스트라고 한다. 프리 리스트의 전체 노드 개수 및 각 노드의 크기는 가변적이지만, 항상 블록의 시작 주소 순으로 연결된다. 이중 링크드 리스트로 구성되는 프리 리스트, 즉 할당 가능한 영역은 그림 1과 같이 할당 요청의 특성에 따라 조절되는 영역 구분 점에 의해 두 개의 영역으로 구분된다.

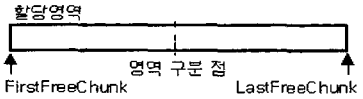

그림 1 할당 영역의 구조

실행 프로그램으로부터 할당 요청을 받은 할당기는 요청크기에 따라 할당 위치를 결정한다. 비교적 작은 할당 요청을 받은 경우 FirstFreeChunk부터, 큰 요청을 받은 경우 LastFreeChunk부터 할당 가능한 첫 번째 블록을 탐색한다. 요청된 크기보다 블록이 큰 경우 블록은 분할되고 나머지 부분은 다음 할당을 위해 프리 리스트에 남게 된다. 이러한 할당 방법은 하나의 할당 시작 위치를 가지는 경우 보다 메모리 단편화 문제를 최소화하여 메모리 낭비를 줄일 수 있도록 유도한다.

3.3 객체의 할당

스탑더월드 방식에서 콜렉터와 실행 프로그램간 도달 가능한 그래프의 일관성은 항상 유지되며 할당기는 자신의 할당정책에 따라 객체를 위한 메모리 공간을 할당한다. 그러나 점진적 가비지 콜렉터는 실행 프로그램과 콜렉터가 번갈아 실행됨에 따라 도달 가능한 그래프의 일관성이 유지되지 않을 수 있다. 다음 그림 2는 전체 메모리 공간에서 마크를 위한 그래프 탐색이 B지점까지 진행된 후 사용자의 프로그램이 실행되면서 객체 A와 C가 할당된 경우이다. 객체 C가 생존기간이 남아 있다면, 그래프 탐색이 진행됨에 따라 마크되어 회수 단계에서 회수되지 않는 것이 보장된다. 그러나, 이미 그래프 탐색이 완료된 지역에 할당된 객체 A는 생존기간에 관계없이 회수 단계에서 콜렉터에 의해 회수된다.

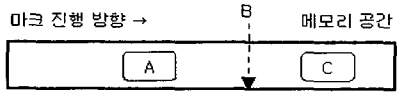

그림 2 콜렉터와 객체의 할당

본 논문에서는 객체 A를 콜렉터로부터 보호하기 위해, 할당기는 콜렉션의 한 주기인 마크 단계의 시작부터 회수 단계 종료시점까지 할당되는 객체를 마크한다. 이로 인하여, 콜렉터의 작업 진행 여부에 상관없이 할당된 객체는 마크된 상태이므로 자신의 생존기간까지 가비지 콜렉터로부터 보호될 수 있다.

3.4 가비지 콜렉터의 시작

기존의 점진적 가비지 콜렉터는 실행 프로그램이 메모리 할당을 요구하거나 포인터 연산을 발생시키는 경우 콜렉션을 수행하는 방식으로 작업을 분할하였다. 그러나 엄격한 실시간 시스템에 비해 대화형 시스템은 비교적 관대한 작업 마감시간을 가지며, 사용자의 다음 행동을 기다리는 시간이 존재한다. 그러므로, 실제 연산이 발생하는 시점에 콜렉션이 함께 실행되는 방식보다는 주기적으로 실행되도록 하였다. 가상머신 내부에서 가비지 콜렉터의 실행은 그림 3과 같이 또 하나의 자바 스레드처럼 실행된다.

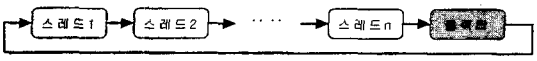

그림 3 가비지 콜렉터의 시작

콜렉터가 실행되는 또 다른 경우는 사용 가능한 공간이 부족하여 메모리 할당에 실패한 경우이다. 이러한 경우 가비지 콜렉터는 기존의 스탑더월드 방식처럼 남아 있는 콜렉션 작업을 모두 실행한다. 그리고 나서 도달 가능한 경로가 존재하지 않음에도 불구하고 회수되지 않은 플로팅 객체(floating object)를 회수하기 위해 콜렉션을 한번 더 실행한다. 이후에도 할당에 실패한다면 자바가상머신은 OutOfMemoryError를 발생시키고, 종료한다.

한가지 고려해야할 점은 실행 프로그램이 메모리 할당을 요구하지 않은 경우에도 가비지 콜렉터가 실행되어야 하는지의 여부이다. 메모리 할당이 발생하지 않았다면 콜렉터에 의해 회수될 가비지도 존재하지 않으므로 실행은 의미가 없다. 그러므로, 콜렉션의 새로운 주기가 시작되기 전에 실행되었던 모든 스레드가 할당을 요구하지 않은 경우에는 콜렉터가 실행되지 않고 다음 스레드가 실행된다.

3.5 실행 정보의 저장과 복구

점진적 콜렉션 방식에서 고려해야할 것 중 한가지는 콜렉터의 상태 정보 유지 방법이다. 본 논문에서는 콜렉터의 실행정보 저장을 위해 네 가지 형태의 변수를 사용한다. 이는 몇 번째 연산이 실행되어야 하는지 나타내

는 collectionKey, 콜렉터의 각 단계 상태 정보를 저장하기 위한 saveMarkInfo, saveChildInfo, saveSweep Address이다.

콜렉터가 시작되면 collectionKey 값에 따라 몇 번째 단계가 시작되어야 하는지 결정하고, 그림 4와 같이 해당 연산의 시작 위치에서 실행정보가 복구된다. 복구된 정보에 따라 콜렉션이 진행되고, 진행 중에 지정된 작업양에 도달했는지 확인한다. 도달했다면 다음 콜렉션이 시작되었을 때 실행되어야 할 위치와 상태 정보를 저장하고 콜렉션은 종료된다. 그렇지 않은 경우 다음 단계를 위해 상태 정보를 초기화한다.

```
void markGlobalRoots() {
    /* 지역 변수의 생성 및 초기화 */
    int i = 0;

    /* 상태 정보의 복구 */
    if(saveMarkInfo.i > 0) i = saveMarkInfo.i;

    /* 라이브 객체 마크 */
    for(; i < length; i++) {
        .....
        if(지정된 양을 초과했는가?) {
            /* 다음번 실행을 위한 정보저장 */
            return;
        }
    }
    /* 다음 단계를 위한 초기화 작업 */
}
```

그림 4 상태 정보의 복구와 저장

3.6 가비지의 회수

마크 단계가 종료된 후 가비지가 점유한 공간을 재활용하기 위한 회수 단계가 시작된다. 기본적인 알고리즘은 2.4절에서 설명된 것과 동일하지만 두 가지 차이점이 있다. 첫 번째는 3.7절에서 설명될 칼라토글방식으로 인해 회수되지 않은 객체의 마크 비트를 삭제하기 위한 연산이 필요하지 않다는 점이다. 두 번째는 회수 단계의 분리 문제를 고려해야 한다는 점이다. 마크 단계와 같이 회수 단계도 분할되므로 메모리 전체가 한번에 회수된다는 보장이 없다. 그러므로 객체의 크기에 따라 할당 위치를 결정하는 할당기의 정책을 지원하기 위해서는 프리 리스트에 포함될 블록의 범위를 고려해야 한다. 본 논문에서는 현재 가비지 콜렉션 주기에서 회수된 블록들과 이전 주기에서 회수된 블록들을 모두 사용한다. 만약 이전 주기에서 회수된 블록과 연결하지 않는다면 할당기가 자신의 정책에 따라 객체를 구분하여 할당하더라도 전체 메모리에서 객체들은 분리되지 않을 것이다.

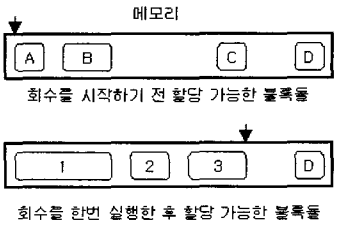

그림 5 회수 단계의 프리 리스트 처리

그림 5는 회수 단계에서 가비지를 회수하는 방법을 나타낸다. 회수 단계는 블록 A의 시작위치에서 시작되었고, 블록 B와의 사이에 위치한 가비지가 회수되어 새로운 블록 1이 생성되었다. 이후 블록 3의 위치까지 모든 가비지가 회수된 후 다시 사용자의 프로그램이 실행되어야 한다면, 다음 번에 회수를 시작할 위치인 블록 3의 다음 번지를 saveSweepAddress에 저장하고, 이전 콜렉션 주기에서 회수되었던 블록 D를 블록 3과 연결하여 프리 리스트를 구성한다. 회수 단계의 종료는 메모리의 가장 마지막 블록을 확인한 후이다.

3.7 마크 비트의 의미 변경

콜렉터는 생존기간이 끝나지 않은 객체는 회수하지 않는다는 것을 보장해야 하며, 생존기간이 끝난 객체는 회수해야 한다. 이를 위해 본 논문에서는 객체의 상태를 마크된 경우와 그렇지 않은 두 가지 상태로 구분하고 콜렉션 주기의 마지막에 마크 비트의 의미를 변경하는 칼라 토클 방식을[15] 사용한다. 그러므로, 회수 단계에서는 객체의 마크 비트를 삭제할 필요가 없다.

그림 6의 (a)는 객체 A에서 도달 가능한 객체들이 마크된 후 콜렉션 작업이 중단된 경우이다. 이후 실행 프로그램에 의해 그림 6의 (b)와 같이 이미 마크된 객체 C는 새로운 객체 F를 지시하도록 변경되었고, 다시 콜렉터가 실행되어 객체 B에서 도달 가능한 모든 객체가 마크되었다. 객체 F는 콜렉션 중에 할당되었으므로 마크된 상태로 할당된다. 회수 단계에서 객체 E는 플로팅 객체로 남는다. 그림 6의 (c)는 회수 단계가 종료되고

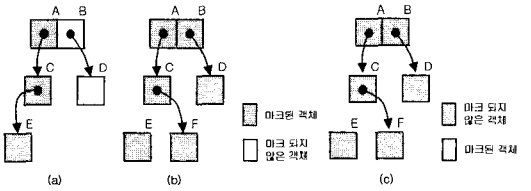

그림 6 플로팅 객체의 처리

마크 비트의 의미를 변경한 이후의 객체 상태를 나타낸다. 모든 객체들은 이전과 동일한 마크 비트를 가지지만 비트의 의미 변경으로 인해, 다른 상태가 되었다. 이후 새로운 콜렉션 주기가 시작되면, 객체 E는 어떠한 경로로도 도달 불가능하므로 마크되지 않을 것이며 회수 단계에서 회수되는 것이 보장된다.

3.8 가비지 콜렉터의 종료

가비지 콜렉터의 한 주기가 종료되기 전에 몇 가지 작업이 실행되어야 한다. 우선, 다음 콜렉션 주기를 위해 실행 정보 저장 공간에 대한 초기화 작업을 한다. 두 번째는 콜렉션 주기가 종료되었음을 알리는 플래그 비트를 설정하여 할당기가 객체를 할당할 때 마크되지 않은 상태로 할당할 수 있도록 해야 한다. 마지막으로 모든 객체를 마크되지 않은 상태로 만들기 위해 마크 비트의 의미를 변경한다. 가비지 콜렉터는 메모리 할당에 실패하지 않는다면 자바가상머신이 실행되는 동안 스레드의 스케줄링 정책에 따라 주기적으로 실행된다.

4. 실험 및 성능 평가

대화형 시스템에서 가비지 콜렉션으로 인해 발생되는 사용자 프로그램의 지연시간은 인식할 수 없을 정도의 작은 시간이라야 한다. 사용자 프로그램이 작은 지연시간을 가진다는 것은 시스템이 사용자의 행동에 대해 보다 신속한 반응을 보인다는 것을 의미한다. 이 장에서는 본 논문에서 제안한 점진적 마크-회수 가비지 콜렉터가 기존 방식보다 작은 지연시간을 제공하는지 확인하기 위하여 콜렉션의 수행으로 인해 발생되는 지연시간을 측정 및 비교하였다.

4.1 실험 환경

실험은 SUN Ultra-60의 SUN OS 5.7에서 SUN의 KVM과 gcc를 사용하여 이루어졌다. 가비지 콜렉터의 성능 평가는 KVM에서 기본으로 제공하는 스탑더월드 방식과 본 논문에서 제안한 점진적 방식의 콜렉터를 KVM의 가비지 콜렉터와 대치하여 실험하는 방식으로 이루어 졌다. 또한, 실험에서 사용된 KVM의 메모리는 전체 실험 프로그램에서 최소한으로 요구되는 크기인 50KB로 제한하였다.

실험 프로그램은 가비지 콜렉터에게 심한 부담을 주도록 의도된 그룹 A와 실험의 일반성을 얻기 위해 가비지 콜렉션이 요구되는 일반 예제 프로그램으로 구성된 그룹 B로 구분된다. 가비지 콜렉션이 거의 발생하지 않는 프로그램은 콜렉션으로 인한 지연 효과를 확인할 수 없으므로 실험에서 배제하였다. 첫 번째 그룹 A는 가비지 콜렉터의 작업량을 최대한 증가시키기 위하여 메모리가 고갈될 때까지 반복적으로 임의의 크기와 생존 기간을 가지는 객체를 할당하는 mem과 메모리 사용 부담이 심한 순열을 구하는 perm이다. 그룹 B의 실험 프로그램으로는 사용자가 로켓을 조정하여 별을 피하는 star와 이벤트에 따라 최대 30개의 공이 움직이는 ball을 선택하였다.

지연시간의 측정은 동일한 실험 프로그램이 실행되는 동안 C언어의 함수를 통해 얻어진 콜렉션의 시작시간과 종료시간을 로그파일에 기록한 후 분석하는 방식으로 이루어졌다. CLDC를 지원하는 장비마다 CPU 성능 차이로 인하여 실제 측정된 시간은 의미가 없으므로 실험 결과에서 그래프의 시간 단위는 생략하였다.

4.2 지연시간의 비교

자바가상머신은 실행 프로그램에서 소비한 메모리의 재생을 위해 프로그램의 실행을 중단하고 가비지 콜렉션을 수행한다. 가비지 콜렉터의 성능 평가를 위해 측정된 실행 프로그램의 지연시간에 대한 의미는 다음 그림 7과 같다.

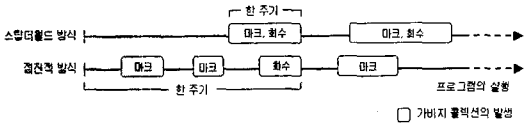

그림 7 지연시간

가비지 콜렉션 발생 시 사용자 프로그램은 지연시간을 가지는데 이러한 지연시간의 간격이 가장 짧은 것이 최소 지연시간이며, 반대의 경우가 최대지연시간이다. 평균 지연시간은 발생된 전체 지연시간의 합을 지연의 발생횟수로 나눈 것이다. 마크 단계부터 회수 단계의 종료시점까지 발생한 전체 지연시간의 합이 한 주기 지연시간이며 한 주기 지연시간에 대한 전체 합을 주기의 개수로 나눈 것이 한 주기 평균 지연시간이다. 스탑더월드 방식은 콜렉션 작업이 분할되어 실행되지 않기 때문에 평균 지연시간과 한 주기 평균 지연시간이 동일하다.

다음 그림 8은 실험 프로그램 중 하나인 mem을 실행하는 과정에서 발생한 가비지 콜렉터의 로그 파일 일

mem의 가비지 콜렉션 로그 파일

	스탑더월드 방식		점진적 방식	
순번	시작시간	종료시간	시작시간	종료시간
1	133.533036	133.533398	973.572313	973.572444
2	133.544548	133.544851	973.572819	973.572876
3	133.553157	133.553564	973.581216	973.581307
4	133.560154	133.560420	973.58291	973.582998
5	133.572161	133.572486	973.583143	973.583203
...	...	...	...	...

그림 8 로그 파일의 내용

부이다. 로그 파일에서 중요한 내용은 콜렉션이 발생했을 때의 지연시간으로 이는 종료시간에서 시작시간을 감소함으로써 얻을 수 있다. 시작 및 종료시간은 가비지 콜렉션 발생 시 결정되는 기준시간에 대한 차이로 표현되므로 콜렉션 발생 간격간의 수치는 의미가 없다.

다음 그림 9는 그림 8과 같은 로그 파일의 내용을 사용하여 4.1에서 설명된 전체 실험 프로그램의 지연시간 평균을 비교한 것이다.

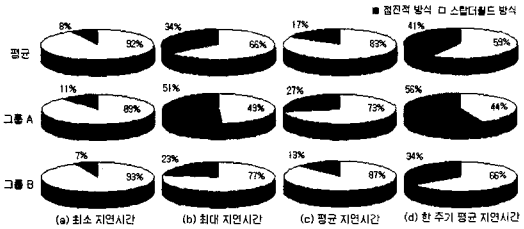

그림 9 콜렉션 방식에 따른 그룹별 지연시간 비교

그림 9의 평균 그래프는 점진적 방식의 가비지 콜렉터가 기존의 스탑더월드 방식보다 작은 지연시간을 제공한다는 점을 나타낸다. 그러나 실험 프로그램이 가비지 콜렉터에게 극심한 부담을 주도록 의도된 그룹 A의 경우 점진적 방식에서의 최대 지연시간이나 한 주기 평균 지연시간이 더 크다. 이는 3.5절에서 설명된 실행 정보의 저장 및 복구 연산의 오버헤드로 인한 것이다. 그룹 B의 한 주기 평균 지연시간은 연산의 오버헤드가 존재함에도 불구하고 스탑더월드 방식보다 작은 지연시간을 가진다. 이는 스탑더월드 방식의 경우 더 이상 할당이 불가능한 시점에 가비지 콜렉션이 시작되므로 한 주기의 작업 공간이 전체 메모리의 크기와 동일하다. 그러나 점진적 방식의 경우 스레드 스케줄링에 따라 주기적으로 분할된 작업이 수행되므로 작업 공간은 콜렉션이 시작되기 전에 스레드에 의해 사용된 공간의 크기에 따르기 때문이다.

4.3 지연시간의 빈도

그림 10은 실험 프로그램이 실행되는 동안 콜렉터에 의해 지연된 시간과 횟수에 대한 그래프이다. 그래프를 통해 콜렉션 발생 시 점진적 방식의 콜렉터가 스탑더월드 방식의 콜렉터 보다 대부분의 경우 작은 시간을 소모하여 더 짧은 지연시간을 가진다는 점을 알 수 있다. 이는 점진적 방식의 콜렉터가 전체 콜렉션 작업을 분할하여 실행하고 또한 한 주기 동안 회수해야할 메모리의 공간이 비교적 작기 때문이다.

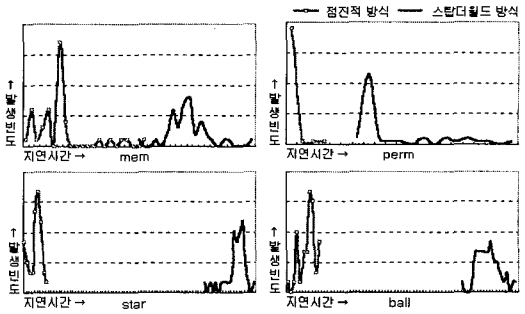

그림 10 지연시간의 빈도

점진적 방식에서 지연시간이 가장 큰 경우는 메모리 공간이 부족하여 메모리 전체를 대상으로 한 주기의 가비지 콜렉션을 한번에 수행해야 하는 경우이다. 이러한 상황에 대한 결과가 mem이며 실험 결과를 통해 점진적 방식의 콜렉터는 최악의 경우 점진적 방식이 가지는 오버헤드로 인해 스탑더월드 방식보다 큰 지연시간을 가지게됨을 알 수 있다.

4.4 프로그램의 실행에 따른 지연시간

사용자 프로그램이 실행됨에 따라 메모리의 사용 가능한 공간은 점점 줄어들고 단편화는 더욱 심각해진다. 결국에는 사용자 프로그램이 요청하는 메모리 공간 확보가 불가능하게 되고 가상머신이 종료된다. 그림 11은 시간에 따라 조건이 달라지는 환경에서도 가비지 콜렉터가 지속적으로 작은 지연시간을 제공할 수 있는지를 확인하기 위한 그래프이다.

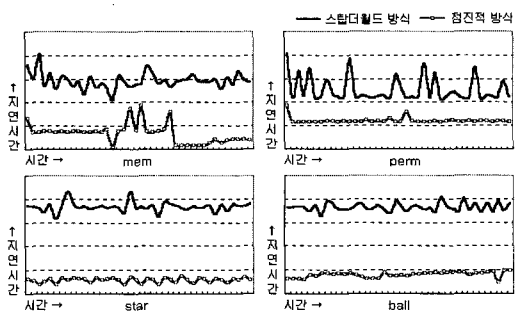

그림 11 프로그램의 실행에 따른 지연시간

그래프를 통해 점진적 방식의 콜렉터가 콜렉션 작업의 분할로 인해 스탑더월드 방식보다 작은 시간을 비교적 일정하게 소모함을 알 수 있다. 이는 점진적 방식의 콜렉터를 사용하는 사용자가 작고 비교적 일정한 지연

시간을 가지게됨을 의미한다.

5. 결 론

CLDC를 지원하는 자바가상머신인 KVM의 가비지 콜렉터는 사용자 프로그램의 실행을 중단하고 가비지 콜렉션을 수행하는 스탑더월드 방식으로 동작한다. 이 방식에서 사용자는 예측할 수 없는 지연시간 동안 가비지 콜렉션이 완료되기를 기다려야 한다. 또한 CLDC에서의 메모리 제한으로 인해 전체 메모리를 분할하여 사용하는 가비지 콜렉션 알고리즘을 사용하기 어렵다.

본 논문에서는 KVM의 메모리 제약과 가비지 콜렉터의 예측할 수 없는 지연시간 문제를 극복하기 위한 가비지 콜렉션 알고리즘을 제안하였다. 개선된 가비지 콜렉터는 메모리를 분할하여 콜렉션 하지 않고, 콜렉션 과정을 분할하여 항상 전체 메모리를 사용할 수 있도록 하였다. 또한, 객체의 크기에 따라 메모리 공간의 할당 위치가 결정되는 양방향 할당 방법을 사용하여 메모리의 효율적 사용을 유도하였다. 그리고, 콜렉션 주기에 따라 마크 비트의 의미를 변경하여 회수 단계에서 전체 라이브 객체의 상태를 하나씩 변경해야하는 문제점을 개선하였다.

실험을 통하여 KVM의 스탑더월드 방식과 본 논문에서 제안한 점진적 방식의 콜렉터를 비교하여 제안된 방식이 다음 사항에서 개선되었음을 확인하였다. 우선, 개선된 가비지 콜렉터는 콜렉션 과정을 분할하여 실행함으로써 사용자에게 작은 지연시간을 제공한다. 또한 시간에 따라 메모리 조건이 달라지더라도 지연시간의 변화 폭은 기본 방식 보다 작았다. 그러나 가비지 콜렉션과 사용자 프로그램의 실행 제어 오버헤드로 인해 한 주기 동안 필요로 하는 시간이 증가되었다. 그러나 이러한 오버헤드는 프로그램이 종료되는 시점까지 고르게 분포되어 사용자에게 미치는 영향이 매우 작다.

오늘날 모든 정보기기와 정보가전 분야에서 자바의 활용도가 급속히 확대되고 있다. 이러한 장치들은 대화형 시스템뿐만 아니라 작업완료 시간이 엄격한 실시간 시스템이기를 요구한다. 이를 위해서는 현재 어느 정도 완화된 시간 제약을 강화해야한다. 따라서 향후 연구에서는 가비지 콜렉터가 엄격한 작업 마감시간을 가질 수 있도록 확장할 것이다.

참 고 문 헌

[1] James Gosling, Henry McGilton, The Java Language Environment a White Paper, Sun Microsystems, 1996.

[2] Sun Microsystems, Java 2 Platform Micro Edition(J2ME) Technology for Creating Mobile Devices White Paper, Sun Microsystems, 2000.

[3] Sun Microsystems, Connected, Limited Device Configuration Specification Version 1.0a Java 2 Platform Micro Edition, Sun Microsystems, 2000.

[4] Tim Lindholm, Frank Yellin, The Java Virtual Machine Specification Second Edition, Addison Wesley, 1999.

[5] Paul. R. Wilson, "Uniprocessor Garbage Collection Techniques," Technical report, University of Texas, Jan 1994. Expanded version of the IWMM92 paper.

[6] William M. Newman, Michael G. Lamming, Interactive System DESIGN, Addison-Wesley, 1995.

[7] Roger Henriksson, Scheduling Garbage Collection in Embedded Systems, PhD thesis, Lund University, 1998.

[8] Perry Cheng, Guy E. Blelloch, "A Parallel, Real-Time Garbage Collector," ACM SIGPLAN Conference on Programming Design and Implementation, pp. 125~136, 2001.

[9] Sun Microsystems, The HotSpot Virtual Machine Technical White Paper, Sun Microsystems, 2001.

[10] Jacob Seligmann, Steffen Grarup, "Incremental Mature Garbage Collection Using the Train Algorithm," In European Conference on Object-Oriented Programming. Springer-Verlag, pp. 235~252, 1995.

[11] Shogo Matsui, Yoshio Tanaka, Atsushi Maeda, Masakazu Nakanishi, "Complementary Garbage Collector," The Transactions of Information Processing Society of Japan, Vol.36, No.8, pp. 1874~1884, 1995.

[12] Frank Yellin, Inside the The K Virtual Machine (KVM), http://servlet.java.sun.com/javaone/javaone2000/pdfs/TS-1507.pdf, 2000.

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

[14] Tuning Garbage Collection with the 1.3.1 Java Virtual Machine, http://java.sun.com/docs/hotspot/gc

[15] Tamar D., Elliot K. K., Ethan L. Eliot E. S., Katherine B., Itai L., Erez P., Igor Y. Yossi L., "Implementing an On-the-fly Garbage Collector for Java," Proceedings of the ACM SIGPLAN Symposium on Memory Management, pp. 155~166, 2000.

권 혜 은

2000년 2월 세명대학교 정보처리학과 졸업. 2003년 ~ 현재 세명대학교 대학원 전산정보학과 석사과정 재학. 관심분야는 프로그래밍 언어

김 상 훈

1989년 동국대학교 대학원 컴퓨터공학과 졸업(공학석사). 1996년 동국대학교 대학원 컴퓨터공학과졸업(공학박사). 1997년 ~ 현재 세명대학교 컴퓨터정보학부 조교수. 관심분야는 객체지향프로그래밍언어, 컴파일러설계및구성, 병행프로그래밍 언어