자바에서 프로파일에 기초한 세대기반 가비지 콜렉터 설계

김일부호, 오세만
동국대학교 컴퓨터공학과

A Design of Profile Based Generational Garbage Collector in Java

Ilbooho Kim, Seman Oh
Dept. of Computer Engineering, Dongguk University

요 약

자바는 자동 메모리 회수(garbage collection) 방식을 채택한 범용 프로그래밍 언어로 자바 가상머신(JVM)이 설치된 다양한 플랫폼에서 사용되어지고 있다. 현재 자바에서 사용되는 가비지 콜렉터는 휴지(pause) 시간이 상당히 길어 짧은 응답시간을 요구하는 서버 및 실시간 응용 프로그램에는 적합하지 않은 표시-압축 기법을 사용한다. 이를 보완하기 위해 자바 HotSpot™ 성능 엔진에서 세대기반(generational) 복사 기법에 기반을 둔 혼성(hybrid) 가비지 콜렉터를 사용하고 있으나, 상당히 큰 오버헤드를 보이고 있어 다양한 응용 프로그램의 특성을 수용하기에는 개선의 여지가 많다.

본 논문에서는 세대기반 기법을 기반으로, 자바와 자바 가상머신이 가진 특성들과 실행시간 객체의 타입 및 스택 정보를 이용하는 가비지 콜렉터를 설계한다. 또한, 힙 프로파일 분석기를 구현하고, 이를 통해 응용 프로그램에 적합한 메모리 구조를 분석하고, 가비지 콜렉터에 적용할 수 있도록 한다

1. 서 론

자동 메모리 관리는 응용프로그램 개발의 생산성 측면에서 높은 효율을 제공한다. 자동 메모리 관리는 메모리 할당정책과 회수정책으로 구분할 수 있으며, 주로 회수정책에 따라 할당정책이 결정된다. 회수정책은 폐기 메모리를 가용 메모리로 환원시키는 방법으로, 안전성과 정확성이 고려되어야 하며, 프로그래머가 신뢰할 수 있어야 한다.

가비지 콜렉터(garbage collector: GC)는 기본적으로 참조계수(reference counting) 기법과 탐색(tracing) 기법으로 구분되며, 탐색 기법은 표시(mark) 기법과 복사(copying) 기법으로 구분된다. 이러한 기본 알고리즘은 서버 및 실시간 응용 프로그램에 적합한 점진적(incremental) 혹은 병렬/분산 특성을 갖는 형태로 응용되며, 응용 프로그램과의 동기화를 위해 읽기/쓰기 경계(barrier)를 사용하기도 한다. 또한 여러 가지 GC 알고리즘을 복합적으로 사용하는 혼성(hybrid) 기법 및 메모리 소비 형태에 따른 적응적(adaptive) 기법으로 응용되기도 하며, 상황에 따라 별도의 하드웨어 지원이 있을 수 있다.[3, 7]

자바는 이러한 자동 메모리 회수 방식을 사용하는 범용 프로그래밍 언어로 다양한 플랫폼에서 사용되고 있으며, GC 또한 해당 플랫폼의 메모리 정책에 기반을 두고 구현되었다. 그러나 다양한 응용프로그램의 특성을 수용하기에는 상당히 긴 응답 시간 및 전반적인 응용 프로그램의 실행 속도 저하와 많은 메모리 사용 등 개선의 여지가 많다.[4, 5]

따라서, 본 논문에서는 세대기반 기법을 기반으로 자바와 JVM이 가진 특성들과 응용 프로그램의 실행시간에 객체의 타입 및 자바 가상머신의 스택 프레임 프로파일을 객체의 세대 및 세대간 루트 셋의 결정에 활용함으로써 탐색 및 복사 시간을 단축한다. 또한, 힙 프로파일 분석기를 구현하고, 이를 통해 응용 프로그램에 적합한 메모리 구조를 분석해 적용함으로써, GC가 다양한 응용 프로그램의 특성을 수용할 수 있도록 한다.

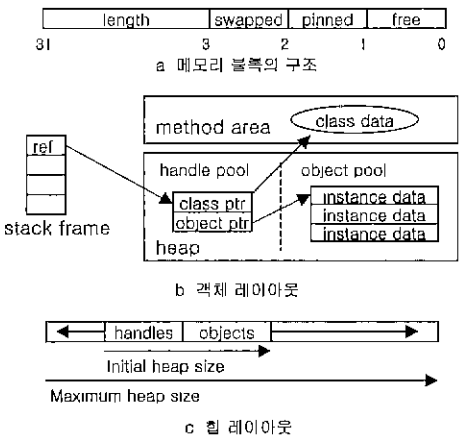
본 논문은 자바에서 사용하는 자동 메모리 관리의 장단점을 2장에서 살펴보고, 3장에서는 관련 연구와 실행 시간 프로파일 정보에 기초한 세대기반 메모리 회수 정책을 설계하며, 4장에서 결론 및 향후 연구 과제를 제시한다.

2. 자바의 메모리 관리

현재 SUN™사의 자바에서 사용하는 GC 알고리즘은 JDK 1.02에서 사용하던 표시-수거(mark-sweep) 기법을 개량한 표시-압축 기법을 사용하고 있다. 표시-수거 기법은 참조 계수

기법의 단점인 순환 참조(cyclic reference) 문제를 해결할 수 있고, 비교적 구현하기 용이한 특성을 갖고 있으나, 일괄처리(stop and go) 방식에서 메모리 회수에 걸리는 시간이 소비된 메모리 크기에 따라 비례하므로 GC로 인해 응용 프로그램의 응답시간 저하를 가져올 수 있고, 메모리 단편화가 발생하는 단점이 있다. 또한 NMI(Native Method Invocation)를 통해 생성된 폐기 메모리의 회수를 위하여 보존적(conservative) 기법을 병행함으로써 정확성이 떨어졌다.[6]

이후 사용된 표시-압축 기법은 폐기 객체의 수집 시 메모리 압축을 고려하여 단편화 문제를 보완했으나, 수집 시간의 오버헤드는 증가했다. 그리고, JDK 1.1에서 소개된 JNI(Java Native Interface)는 네이티브(native) 객체를 자바 객체와 동일하게 취급할 수 있도록 별도의 핸들(handle)을 사용함으로써 네이티브 객체에 대한 보존성 문제를 보완하고, 정확성을 높일 수 있었다. 위와 같은 폐기 메모리 회수 정책에 따라 SUN™사의 JVM은 [그림 1]과 같은 형태로 힙 영역에 객체를 관리한다.[6]

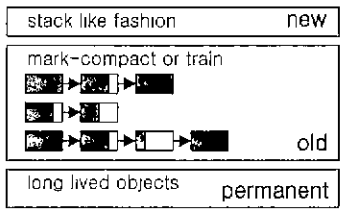


[그림 1] JDK 1.2에서 자바의 메모리 관리

표기-압축 기법은 GC가 실행되는 동안 응용 프로그램의 휴지(pause) 시간이 상당히 길어 서버환경에는 적합하지 않기 때문에, SUN™사는 자바 HotSpot™ 성능 엔진에서 세대별 복사(generational copying) 기법과 점진이며 휴지 시간이 짧은 Train 기법[5]을 추가 적용했다. 세대별 복사 기법은 객체의 수명에 따라 분리된 각 세대별로 메모리 회수 빈도를 차별화함으로써 휴지 시간을 줄일 수 있다.[4]

HotSpot™에서 자바의 힙은 글로벌 성격의 객체를 위한 permanent 세대, 수명이 긴 객체를 위한 old 세대 그리고 nursery라 불리는 스택 형태의 new 세대로 구분되며, old 세대와 new 세대간에 복사 기법을 적용해 단편화 문제를 해결했다. 그러나 응용 프로그램의 수행 시간이 길어질 경우 old 세대의 크기가 커지고, 휴지 시간 또한 상대적으로 증가한다. 이러한 경우 표기-압축 기법이 기본적으로 사용되며, 선택적으로 Train 기법이 적용될 수 있다.[4]

Train 기법은 old 세대의 객체들을 car라 불리는 고정된 메모리 블록에 할당하고, 이 car들을 참조 지역성(locality)에 따라 특정 train에 연결시킨 후, GC의 적용을 car 별로 차별화함으로써 휴지 시간을 줄이는 기법이다. 그러나 복잡한 메모리 구분과 할당으로 HotSpot™에서 약 10% 정도의 오버헤드를 보이고 있다. 다음 [그림 2]는 자바 HotSpot™ 성능 엔진에서 사용하는 힙 레이아웃을 보이고 있다.[4, 5]



[그림 2] HotSpot™의 힙 레이아웃

또한 자바는 객체의 참조 관계를 제한된 방법으로 명시할 수 있는 참조(Reference) 객체 API를 제공한다. 참조 객체는 soft 참조 객체, weak 참조 객체 그리고 phantom 참조 객체로 구분되고, 루트 셋으로부터의 참조 도달 가능성에 따라 GC와 상호 작용을 가능케 하며, 일시적으로 많은 메모리를 사용하는 객체들에 적합하다. 그리고 자바의 객체는 객체의 소멸 시 반드시 수행되어야할 코드를 위해 Finalizer를 가질 수 있다. Finalizer는 객체가 제거되기 전에 반드시 한번만 수행되어야 하며, 본래의 목적에서 벗어나 다른 용도로 사용할 경우 객체 소생(resurrect) 문제가 발생할 수 있다.[6]

3. 프로파일 기반의 세대별 가비지 콜렉터

3.1 관련 연구

프로파일 정보를 GC에 이용하는 방법은 컴파일 시간에 분석 수집된 정보를 이용하는 컴파일 시간 GC와 실행 시간에 필요한 정보를 수집해 이용하는 tagless GC, 그리고 실험적 수행을 통해 얻은 결과를 이용하는 방법으로 구분할 수 있다.

컴파일 시간 GC는 주로 함수형 언어에서 closure를 통한 객체의 수명 예측 결과를 이용하며, 사용자와 상호작용으로 인한 객체의 수명 변화 등을 고려한 상당히 보존적 기법에 속한다. 그리고 tagless GC는 실행시간 객체의 타입 정보와 스택의 변화에 대한 프로파일 등을 사용하지만, 타입 분석에 소요되는 시간과 스택의 변화를 기록하는 방법 등 많은 제약이 있다.[1, 2]

3.2 세대기반 가비지 콜렉터

세대기반 기법은 대부분의 객체들이 비교적 짧은 수명을 가지며, 생성된 객체의 일부분만 집중적으로 사용되는 특징을 이용한다. 세대기반 기법은 탐색과 복사 기법을 응용한 방법으로 복사에 소요되는 비용을 줄일 수 있으며, 메모리 단편화를 예방할 수 있고, 휴지시간이 비교적 짧다.[7]

세대기반 기법에서는 세대의 수와 크기 그리고 각 세대의 메모리 회수 빈도에 따라 GC의 성능이 좌우된다. 또한, 각

세대에 관련된 루트 셋을 별도로 관리함으로써 탐색 비용 또한 줄일 수 있다. 반면 한 세대가 지나치게 커질 경우 탐색 및 복사 시간은 증가하며, 이로 인한 휴지 시간은 길어지게 된다.[2, 3, 4]

따라서, 본 논문에서는 응용 프로그램이 사용할 수 있는 힙의 크기를 정하고, 세대의 수에 따라 고정된 크기의 세대를 사용함으로써 프로그램의 특성에 따른 GC의 수행 결과를 얻을 수 있도록 한다. 또한 새로 생성된 객체들이 할당되는 nursery의 크기는 GC의 발생 빈도를 결정할 수 있고, 이 크기에 따라 휴지 시간은 조절될 수 있다.

3.3 Stack 프로파일

자바의 실행 환경은 스택을 기반으로 스레드의 생성 시 하나의 스택을 생성하고, 메소드의 호출이 있을 경우 하나의 스택 프레임을 생성한다. 이러한 스택 프레임은 세대기반 기법에서 힙에 할당된 객체의 세대와 수명을 결정하는 루트 셋으로 사용되며, 각 세대와 관련된 루트 셋만을 별도로 관리함으로서 탐색 비용을 줄일 수 있다.[2]

본 논문에서는 각 세대가 생성될 때 해당 세대에 대한 스택과 스택 프레임 정보를 별도의 메모리 영역에 관리함으로써 탐색 시간을 줄이고, 스택 프레임의 수명에 따라 새로운 세대에 복사될 객체를 결정한다. 그림 3에서 스택 맵은 세대별로 스택을 생성한 스레드의 형 정보, 스택 프레임의 ID 및 caller 메소드의 스택 프레임 ID 등을 관리한다.

3.4 객체의 생존성(liveness) 프로파일

JVM은 메소드 영역에 클래스 데이터를 저장하고, 클래스의 인스턴스는 힙에 저장한다. 자바에서 객체의 타입은 스택 프레임으로부터 메소드 영역의 클래스 정보와 constant pool 을 역 추적해 얻을 수 있고, 객체의 수명을 결정하는데 이용할 수 있다. 그러나 생성된 모든 객체의 형 관리 및 분석에는 상당한 오버헤드가 있다.[1]

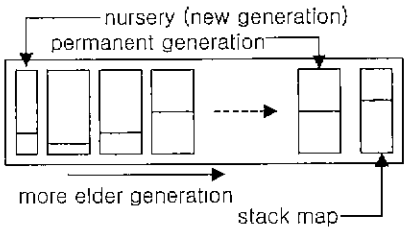
자바는 기본 실행 단위로 스레드를 사용하며, 데몬 스레드를 제외한 모든 스레드가 종료될 때 응용 프로그램의 실행은 종료된다. 따라서 절제된 객체의 형 관리를 위해 스레드 객체의 타입 정보만 스택 맵에 관리한다.

3.5 힙 프로파일 분석기

GC는 다양한 응용 프로그램의 특성을 수용할 수 있어야 하며, 이를 위해 실행 시간 메모리 사용 특성을 분석해 최적의 성능을 얻도록 GC를 제어할 수 있는 방법이 제시되어야 한다. 힙 프로파일 분석기는 스택과 스택 프레임의 생성 및 소멸 그리고 각 세대별 메모리 회수 빈도 등을 분석하고, GC의 폐기 객체 회수 비용을 산출 할 수 있도록 한다.

3.6 메모리 관리

[그림 3]은 전체적인 힙의 구조를 보이고 있다. 사용자는 응용 프로그램을 위한 힙의 크기와 nursery의 크기 및 세대의 수를 지정할 수 있다. 또한 글로벌 성격의 객체를 별도의 영역에 관리함으로써 불필요한 탐색 및 복사 시간을 줄이도록 한다.



[그림 3] 제안한 세대기반 힙 레이아웃

Nursery를 제외한 각 세대는 고정된 크기를 사용하며, 더 이상 새로운 객체를 할당할 공간이 부족할 경우 새로운 세대를 생성하고 스택 맵의 정보를 참조해 복사한다.

4. 결론 및 향후 연구

자바는 범용 프로그래밍 언어로 다양한 응용 프로그램의 특성을 수용할 수 있어야 한다. 이를 위해 자바의 GC는 작고 견고해야 하며, 휴지시간이 짧아야한다. 또한 개발자가 GC를 신뢰할 수 있도록 안정성과 정확성을 보장해야 한다.

본 논문에서는 실행시간에 절제된 객체의 타입 분석과 스택 및 스택 프레임의 프로파일을 이용하고, 힙 프로파일 분석기를 통해 각 응용 프로그램의 메모리 사용 특성을 파악한 후, GC의 동작을 제어할 수 있는 프로파일에 기초한 세대기반 가비지 콜렉터를 설계했고 구현 중이다.

자바는 플랫폼에 독립적인 특성으로, 네트웍 환경 및 분산 환경에서 널리 쓰이고 있으며, 자바의 객체들은 네트웍을 통해 이동 가능하다 이를 위해 분산 가비지 콜렉터 및 실시간 환경에 적합하도록 concurrent GC에 관한 연구가 필요하다.

[참고문헌]

[1] Ole Agesen, David Detlefs, Eliot Moss, "Garbage Collection and Local Variable Type-Precision and Liveness in Java Virtual Machines", ACM SIGPLAN Conference, 1998.

[2] Perry Cheng, Robert Harper, Peter Lee, "Generational Stack Collection and Profile-Driven Pretenuring", Carnegie Mellon University, 1998.

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

[4] Steve Meloan, "The Java HotSpot™ Performance Engine. An In-Depth Look", Sun Microsystems, 1999.

[5] Jacob Seligmann, Steffen Grarup, "Incremental Mature Garbage Collection Using the Train Algorithm", Proceedings of ECOOP, 1995.

[6] Bill Venners, "Inside the Java virtual machine", McGraw-Hill, 1998.

[7] Paul R. Wilson, "Uniprocessor Garbage Collection Techniques", University of Texas, 1992