EVM에서의 자바 동적 메모리 관리기 및 쓰레기 수집기의 구현 및 성능 분석

이 상 윤[†] · 원 희 선[†] · 최 병 욱[†††]

요 약

자바 언어는 객체지향성, 안전성, 유연성으로 인하여 현재 가장 널리 쓰이는 프로그래밍 언어의 하나가 되었으며, 자바 가상머신이 제공해주는 메모리 관리기 및 가비지 컬렉터로 인하여 프로그래머는 메모리 관리에 관한 많은 고민이 줄어들었다. 본 논문에서는 임베디드용 자바 가상머신에서 구현된 메모리 관리기 및 가비지 컬렉터에 대해서 제안한다. 메모리 관리기는 힙을 다양한 크기의 셀로 분할한 후 동일한 셀의 집단인 블록 단위로 관리하여 빠른 메모리 할당과 해제가 가능하다. 가비지 컬렉션 방법으로는 3-색상 기반 표기-쓸어담기 가비지 컬렉터를 기반 알고리즘으로 채택하였으며 멀티쓰레드를 지원하기 위해 새로운 4-색상 기반 표기-쓸어 담기 가비지 컬렉터를 제안한다. 제안하는 기법은 메모리 단편화가 발생하지만 객체 생성수가 많아짐에 따라 단편화율이 거의 일정함을 실험을 통해 보인다.

키워드 : 자바, 가비지 컬렉터, 동적 메모리 관리

Implementation and Performance Analysis of the EVM's Java Dynamic Memory Manager and Garbage Collector

Sang-Yun Lee[†] · Hee-Sun Won[†] · Byung-Uk Choi[†††]

ABSTRACT

Java has been established as one of the most widely-used languages owing to its support of object-oriented concepts, safety, and flexibility. Garbage collection in the Java virtual machine is a core component that relieves application programmers of difficulties related to memory management. In this paper, we propose a memory manager and a garbage collector that is implemented on a embedded java virtual machine. The memory manager divide a heap into various-sized cells and manages it as blocks of same-sized cells. So it is possible to allocate and free memory fast. We adopted the 3-color based Mark & Sweep garbage collector as our base algorithm and we propose 4-color based Mark & Sweep garbage collector for supporting multi-threaded program. The proposed garbage collector occurs memory fragmentation but we show through a experiment that the fragmentation ratio is almost fixed according as we create objects continuously.

Key Words : Java, Garbage Collector, Dynamic Memory Management

1. 서 론

자바(Java) 언어는 그 객체지향성, 안전성, 유연성으로 인하여 현재 가장 널리 쓰이는 프로그래밍 언어의 하나이다. 이러한 자바 언어로 작성된 응용 프로그램이 동작하기 위한 자바 플랫폼은 타깃 시스템의 메모리, CPU 성능, 전력, 응용 프로그램에 적합한 환경 구성을 위해 컨피규레이션을 정의한다[1]. 컨피규레이션은 자바 가상 머신과 코어 API를 정의하고 있으며, J2EE/J2SE/J2ME로 구분된다. 이 중 J2ME(Java2 Micro Edition)는 임베디드 환경을 위한 자바 기술로서 스마트폰, 핸드헬드 디바이스, PDA, 스크린폰, 셋탑 박스, 넷 TV와 같이 네트워크로 연결된 임베디드 혹은 모바일 기기에서 사용되며, 기기의 환경에 따라 다시 CLDC (connected limited device configuration)와 CDC(connected device configuration) 플랫폼으로 구분된다[1]. CLDC 플랫폼은 휴대폰이나 PDA와 같이 메모리나 CPU 성능이 극히 제한적인 기기에 탑재되어 사용되며, CDC 플랫폼은 홈 네트워크, 텔레매틱스, 셋탑 박스 등 좀 더 많은 리소스와 높은 성능을 갖춘 임베디드 시스템에서 주로 사용된다. 자바 언어로 작성된 소스코드는 컴파일되면 자바 바이트코드로 변환된다. 이러한 자바 바이트코드는 자바 가상머신에 의해

† 정 회 원 : 한국전자통신연구원 임베디드S/W연구단 선임연구원
†† 정 회 원 : 한양대학교 정보통신대학 정보통신학부 교수
논문접수 : 2006년 3월 16일, 심사완료 : 2006년 6월 14일

해석되고 실행된다. 즉, 자바 가상머신이란 컴파일된 자바 바이트코드와 실제로 프로그램의 명령어를 실행하는 마이크로프로세서 또는 하드웨어 플랫폼 간에 인터페이스 역할을 담당하는 소프트웨어를 의미한다[2]. 자바 가상머신으로 인하여 한 번 작성된 자바 바이트코드는 자바 가상머신이 동작하는 모든 플랫폼에서 동작이 가능해진다.

자바 가상머신이 가지는 중요한 모듈 중 하나인 쓰레기 수집기(garbage collector)는 응용 프로그램이 더 이상 사용하지 않는 메모리상의 객체를 찾아서 회수하는 역할을 담당한다. 이 결과, 가비지 컬렉터는 프로그래머에게 메모리 관리에 관한 많은 고민을 줄여주었으며, 응용 프로그램이 보다 안정적으로 동작할 수 있게 해준다[3, 4]. 현재, 한국전자통신연구원에서는 임베디드 자바 가상머신인 EVM(Embedded Java Virtual Machine)을 개발하고 있으며 본 논문에서는 EVM에서 구현된 메모리 관리기 및 가비지 컬렉터에 대해서 제안한다.

임베디드 환경은 다음과 같은 특성을 갖는다. 첫째, 대부분의 임베디드 기기들은 메모리 용량이 작다. 둘째, 배터리로 동작되는 경우가 많으므로 전력 소비가 중요한 최적화 척도이다. 셋째, 임베디드 기기들은 일반 데스크탑 기기와 비교하여 CPU의 성능이 떨어진다. 이러한 고유의 특성들을 고려한 임베디드 자바 가상머신은 제한된 메모리의 임베디드 시스템에서 오랜 시간동안 동작하도록 요구됨에 따라 자바 객체를 효율적으로 생성하고 관리하는 것이 매우 중요하다. 즉, 동적 메모리 할당이나 가비지 컬렉션이 자바 가상머신의 성능에 매우 중요한 영향을 미친다[5, 6].

제안하는 메모리 관리기는 메모리를 블록과 셀로 분할한다. 셀은 일정 크기를 갖는 메모리이며 객체 할당 및 가비지 회수의 단위가 된다. 블록은 동일한 크기를 갖는 셀들을 링크드 리스트로 엮어서 관리한다. 메모리를 블록과 셀로 분할함으로써 객체 할당 요구 시 그 크기에 맞는 셀을 빨리 찾음으로 해서 신속한 메모리 할당이 가능해진다. 이는 Kaffe에서도 채택된 방식이다[7]. 제안하는 가비지 컬렉션 방법은 3-색상 기반 표기-쓸어 담기 방식을 기반으로 한다. 이 방법은 Kaffe JVM, Blackdown JVM, Netscape Enterprise Server 등의 시스템에서 채택된 방식이다[8]. 그런데, 3-색상 기반 표기-쓸어 담기 가비지 컬렉터는 단일 쓰레드에서는 아무런 문제없이 동작하지만 자바처럼 멀티쓰레드를 지원하는 플랫폼에서는 다른 쓰레드에서 할당 받은 객체가 루트셋으로 정의되기 이전에 가비지가 컬렉터가 동작되면 그 객체를 가비지로 취급함으로써 가비지 컬렉터 수행 후 그 쓰레드가 재가동 될 때 메모리를 잃어버리는 문제점이 있다. 본 논문에서는 이러한 문제점을 해결하기 위해 4-색상 기반의 표기-쓸어 담기 가비지 컬렉터를 제안한다.

본 논문의 구성은 다음과 같다. 제 2장에서는 메모리 관리기 및 가비지 컬렉터의 관련 연구에 대해서 설명한다. 제 3장에서는 본 논문에서 제안한 동적 메모리 관리기의 메모리 구조 설계와 메모리 맵에 대해서 설명한다. 제 4장에서는 본 논문에서 제안하는 가비지 컬렉션 방법과 루트셋에 대해서 설명한다. 제 5장에서는 EVM에 구현된 메모리 관리기 및 가비지 컬렉터의 실험 결과를 보인다. 마지막으로, 제 6장에서는 본 논문을 요약하고 결론을 내린다.

2. 관련 연구

2.1 동적 메모리 관리기

동적 메모리 관리기는 자바 가상 머신 혹은 자바 프로그램에서 요구하는 메모리를 할당하고 관리한다. 메모리 할당 방식에는 인접 메모리 할당 방식과 자유 리스트 메모리 할당 방식이 있다. 인접 메모리 할당 방식은 전체 메모리를 하나의 연속된 메모리 공간으로 간주하고 새로운 객체를 할당할 때는 객체의 크기만큼 포인터를 증가시킴으로써 새 객체를 연속된 공간의 끝에 인접시키는 방식이다. 자유 리스트 메모리 할당 방식은 메모리를 k byte의 크기로 분할 한 후 자유 리스트로 이들을 관리한다. 자유 리스트는 인접한 메모리의 블록들로 구성되어 있으며 객체를 저장할 때는 저장할 수 있는 가장 작은 크기의 셀에 객체를 할당한다[8].

Kaffe는 메모리를 블록과 셀로 분할하여 관리하는 자유 리스트 메모리 할당 방식을 채택하고 있으며[7] SUN의 CVM은 인접 메모리 할당 방식을 채택하고 있다[10]. 동적 메모리 관리기는 가비지 컬렉션 알고리즘에 따라서 의존하여 운영될 수도 있고 독립적으로 운영될 수도 있다.

2.2 가비지 컬렉터

2.2.1 참조 세기(Reference counting)

참조 세기는 각각의 객체를 참조하는 참조 수를 가비지의 판단 근거로 연관시키는데 그 객체를 접근할 수 있는 객체 수를 기록한다. 보통 참조하는 객체가 새로 발생하면 참조 수를 1 증가 시키고 참조하는 객체가 소멸되면 참조 수를 1 감소시킨다. 참조 수가 0이 되면 해당하는 객체는 가비지로 취급하여 회수한다[11].

참조 세기 방법의 장점은 첫째, 가비지 컬렉션을 위한 메모리 관리 부하가 프로그램 실행 중에 분산된다는 것이다. 참조 세기 방식은 상호 작용이 많거나 실시간 시스템처럼 부드러운 응답 시간이 중요한 시스템에 적합하다. 둘째, 공간적인 참조의 지역성이다. 즉, 참조 세기가 0이되는 객체는 힙 상의 다른 페이지를 조사하지 않고 바로 회수될 수 있다. 셋째, 구현이 쉽다.

참조 세기 방법의 단점은 첫째, 포인터가 덧쓰여질 때마다 참조 수가 조정되어야 하므로 참조 수를 갱신하기 위한 연산에 대한 부하가 높다는 것이다. 두번째는 참조 수를 저장하기 위해 별도의 공간이 사용된다. 세번째는 순환 구조의 가비지에 대한 회수가 어렵다.

2.2.2 표기-쓸어담기(Mark & Sweep)

표기-쓸어담기 방식은 표기와 쓸어담기 라는 두 단계를 거친다. 표기 단계는 루트 셋으로부터 객체 참조 그래프를 통해 직접적으로 혹은 간접적으로 도달가능한 객체를 식별

하고 이를 표기한다[12]. 쓸어담기 단계에서는 루트셋으로부터 도달가능하지 않은 객체들을 결정하고 가비지를 시스템에 반환한다. 이 후, 가비지 및 생존한 객체들을 관리하는 내부 데이터 구조가 갱신된다.

표기 단계를 위해 필요한 시간은 객체 참조 그래프의 크기에 따라 달라질 수 있어서 표기하는데 소비되는 시간은 비결정적이다. 따라서, 실시간성이 요구되는 시스템에는 부적합한 방식이다.

표기-쓸어담기 방식은 참조 세기 방법보다 2가지의 장점이 있다. 첫번째는 순환 구조의 가비지 회수가 가능하다. 두번째는 포인터 연산에 대한 부하가 없다. 단점은 첫번째, 가비지 컬렉터가 동작하는 동안 프로그램은 일시 정지한다. 두번째는 표기 단계동안 모든 살아있는 셀들을 방문하고 쓸어담기 단계에서는 모든 셀을 검사해야 하므로 가비지 컬렉션의 비용이 높다. 세번째는 메모리 단편화를 유발시킨다.

2.2.3 복제(Copying, Semi-space)

복제 컬렉터는 힙을 두개의 반-공간(semi-spaces)으로 동일하게 나눈다(Fromspace, Tospace)[13]. 그 중 하나는 현재 데이터를 담고 있고 다른 하나는 가비지 데이터를 담고 있다. 복제 컬렉터는 Fromspace에 있는 살아 있는 데이터를 모두 추적한 후 살아 있는 객체를 새로운 반-공간인 Tospace로 복제한다. 복제가 모두 끝나면 멈추었던 프로그램은 다시 동작하고 다음 번의 가비지 컬렉션을 위해 두 반-공간의 역할을 뒤바꾼다.

복제 가비지 컬렉션의 장점은 첫번째 메모리 할당 비용이 극히 낮다. 두번째 살아 있는 객체를 Tospace의 바닥에 압축함으로써 메모리 단편화가 제거된다. 단점은 첫번째 두개의 반-공간을 사용함으로써 주소 공간이 두 배로 필요하다. 두번째로 가비지 컬렉션이 수행되는 동안 프로그램은 일시 정지한다. 세번째로 오랜 시간 동안 살아 남는 객체는 반복적으로 복제된다

2.2.4 세대 기반 컬렉션(Generational collection)

세대 기반 컬렉터는 객체들의 나이에 근거하여 힙을 두개 혹은 더 많은 그룹으로 나눈다. 객체들은 처음에 가장 젊은 세대에서 할당된다. 그리고, 오랜 시간 동안 충분히 살아 남으면 더 늙은 세대로 세대 이동이 된다. "대부분의 객체들은 젊어서 죽는다"[9]는 전제를 근거로 출발한 세대 기반 방법은 대부분의 가비지 회수 공간이 발견되는 곳이 젊은 세대이기 때문에 가장 젊은 세대에서 메모리를 회수하도록 노력을 집중한다[14]. 전체 힙을 회수하기 위해서는 종종 그러나 긴 멈춤이 있는 반면에 가장 젊은 세대는 좀 더 자주 회수된다. 가장 젊은 세대가 작기 때문에 멈춤 시간은 상대적으로 짧다.

초기 사망률 때문에 젊은 세대의 객체들을 위한 가비지 컬렉터는 표기 노력이 얼마 들지 않는다. 반면에 늙은 세대에 있는 장수하는 객체들에 있어서의 가비지 컬렉션은 전체 표기가 여전히 필요하다. 세대 기반 컬렉터는 오랫동안 살아 남는 객체를 계속해서 복제해 줘야 하는 단점이 있다. 하지만, 객체들을 나이에 의해 세대별로 묶고 늙은 세대보다 젊은 세대들에 대해서 가비지 컬렉션 회수를 좀 더 많이 함으로써 이러한 단점을 극복하는 것이 가능하다.

3. 동적 메모리 관리기

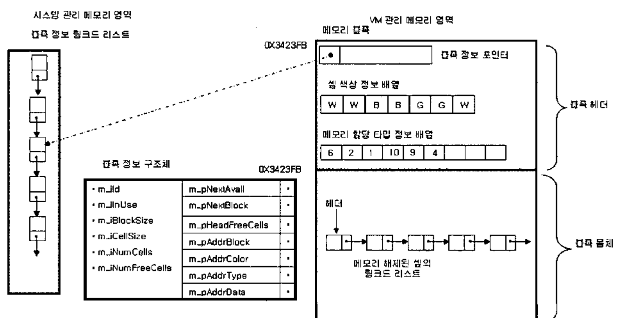
이 장에서는 EVM에 구현된 동적 메모리 관리기에 대해서 기술한다.

3.1 메모리 구조 설계

메모리 구조를 설계함에 있어 우리는 다음을 고려하였다. 첫째, 메모리 할당과 해제가 빨라야 한다. 둘째, 임베디드 시스템 성격상 메모리가 제한적이므로 임베디드 시스템에서 동작하는 프로그램들은 메모리가 작은 객체 중심으로 프로그래밍된다. 따라서, 이에 대한 배려를 해야 한다. 셋째, 큰 객체에 대해서는 별도로 처리 할 수 있어야 한다. 넷째, 메모리 단편화에 대한 고려를 해야 한다. 다섯째, 메모리 용량이 확장 가능해야 한다.

이를 만족하기 위해서 본 논문에서는 자유-리스트 방식을 채택하였다. 인접 메모리 방식은 메모리 단편화율을 줄일 수 있지만, 객체의 크기를 고려하지 않은 방식이며 쓰레기 수집기가 작동한 후 쓰레기를 회수할 때 회수 시간이 자유-리스트 방식보다 길어지므로 메모리 할당 및 해제 시간이 자유-리스트 방식보다 길다. 자유-리스트 방식은 단편화가 발생하는 단점이 있지만, 크기별로 분할된 자유 리스트에서 필요한 크기에 맞는 메모리 공간을 얻어 오기 때문에 메모리 할당이 빠르고 분할된 셀들이 링크드 리스트로 연결되어 있어 메모리 해제를 빠르게 할 수 있다. 또한 메모리 크기를 작은 크기로 세분화 하여 링크드 리스트로 관리하기 때문에 작은 객체들을 쉽게 관리할 수 있다. 또한, 메모리 확장이 용이하다. (그림 1)은 본 논문에서 제안하는 블록의 내부 구조이다.

메모리 영역은 시스템 관리 메모리 영역과 VM 관리 메모리 영역으로 크게 둘로 나뉜다. 시스템 관리 메모리 영역에서는 malloc(), calloc() 과 같은 시스템 호출을 통해 메모리를 얻어 온다. 이 영역은 VM 관리 메모리 영역에서 관리하는 전체 힙을 시스템으로부터 획득하거나 VM 관리 메모



(그림 1) 블록의 내부 구조

〈표 1〉 블록 정보 구조체의 필드

필드명	의미
m_iId	블록의 식별자이며 블록이 생성될 때마다 자동 증가한다
m_pNextAvail	링크드 리스트로 연결된 다음 번의 블록 정보 구조체를 가리킨다
m_pHeadFreeCells	데이터 영역 내의 메모리 해제된 셀로 구성된 링크드 리스트의 헤더를 가리킨다
m_iInUse	블록을 사용하고 있는지를 나타낸다
m_iBlockSize	블록의 크기
m_iCellSize	블록 내의 셀의 크기
m_iNumCells	블록 내의 Cell 개수
m_iNumFreeCells	블록내에서 할당이 가능한 셀의 수
m_pAddrBlock	블록의 시작 주소
m_pAddrType	블록에 속한 셀의 타입 정보
m_pAddrColor	가비지 컬렉션을 하기 위한 셀의 색상 정보
m_pAddrCellInfo	블록에 속한 셀의 정보

리 영역을 제어하는 객체들을 저장하기 위해 사용된다. 그리고, VM 관리 메모리 영역은 가상 머신이 동적으로 객체를 할당하고 해제하면서 관리하는 영역이다. 시스템 관리 메모리 영역에는 블록 정보들이 링크드 리스트로 연결되어 있다. VM 관리 메모리 영역에는 메모리 블록이 존재하는데 메모리 블록은 블록 헤더와 블록 몸체로 나뉜다. 블록 헤더에는 블록 정보 구조체에 대한 포인터와 그 블록에 속한 셀들의 색상 정보와 메모리 할당 타입 정보가 저장되어 있다. 블록 몸체는 셀들이 링크드 리스트로 연결되어 관리된다. 블록은 자바 가상 머신이 기동할 때 설정하는 초기 힙 크기에 따라 블록의 수가 결정되며, 자바 가상 머신의 초기화 과정에서 생성된다. 메모리 할당 요구가 초기로 잡은 힙의 최대 크기를 넘어서면 추가적으로 시스템으로부터 메모리 할당을 받는다.

자바 가상 머신이 기동할 때 시스템으로부터 일정량의 메모리를 할당 받은 후 이 메모리를 블록과 셀로 재구성하여 사용한다. 메모리 할당 타입이 다양한 것은 메모리 관리기가 자바 응용 프로그램에서 요구하는 객체 뿐만 아니라, 가상 머신 엔진에서 요구하는 동적 메모리 할당도 담당하기 때문이다. 이처럼 메모리 관리기가 가상 머신 엔진을 포함하여 전체 힙을 관리하면서 객체 생성과 해제를 담당하기 때문에 메모리 할당과 해제가 빠르며 가상 머신 기동을 빠르게 할 수 있다.

블록은 8KB의 n배로 구성된 메모리 공간으로서 블록에 대한 정보를 나타내는 Block_Info와 셀로 구성되어 있다. 셀의 크기는 16, 24, 32, 40, 48, 56, 64, 80, 96, 112, 128, 160, 192, 224, 240, 496, 1000, 2016, 4040 중 하나이며, 4040 보다 클 경우 8192 의 n 배로 할당된다.

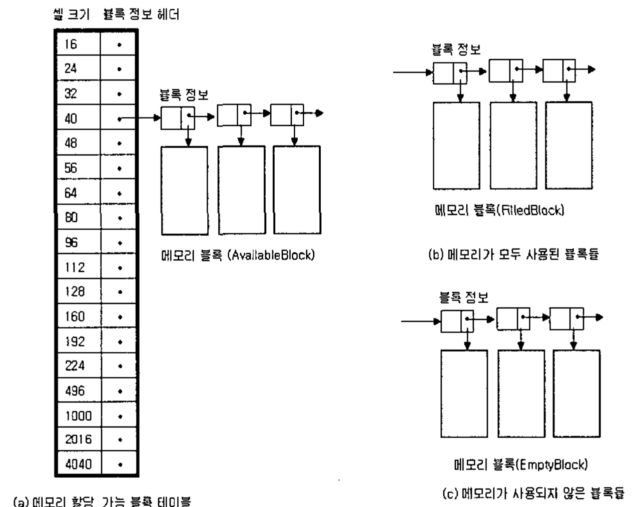
분할하는 블록 및 셀의 크기에 따라, 메모리 할당 및 해제 시간과 메모리 단편화율에 차이가 날 수 있다. 본 논문에서는 임베디드 시스템에서는 작은 객체들이 주로 생성될 것을 감안하여 위와 같이 셀의 크기를 세분화하였다. 블록과 셀은 (그림 1)에서 볼 수 있듯이 단방향 링크드 리스트로 연결되어 관리된다.

3.3 메모리 맵

메모리 맵은 블럭들이 모여 있는 메모리 공간을 관리 하는 역할을 한다. 메모리 맵은 (그림 2)과 같다.

(그림 2) 처럼 블록은 4,040byte 보다 작으면서 사용 할 수 있는 셀을 보유 하고 있는 메모리 블록(AvailableBlock)과 가비지 컬렉터에 의해 모든 블록의 셀이 해제된 블록(EmptyBlock), 그리고 셀들이 모두 사용되어 더 이상 사용할 수 있는 셀이 없는 블록(FilledBlock)들로 나누어져 있다.

(그림 2-a)의 메모리 할당 가능 블록 테이블은 셀 크기별로 해당 블록의 헤더 정보를 갖고 있다. 특정 크기의 메모리 할당 요구가 들어오면 AvailableBlock에서 사용이 가능한 블록을 찾은 후 메모리 할당할 셀을 지정해 준다. 만일 AvailableBlock에서 셀을 구하지 못하면 EmptyBlock에서 새로운 셀을 할당받거나 여기서도 할당 받지 못하면 블록과 셀을 새로 생성한 후 이곳으로부터 할당받는다. 4040 보다 큰 셀을 요구받을 때는 EmptyBlock에서 할당받는다.



(그림 2) 메모리 맵

3.4 객체 생성

메모리 관리기에서 생성하는 객체는 기본적으로 자바 프로그램에서 new 라는 키워드를 통해 생성하는 객체를 생성하는 역할을 하지만, 본 논문에서 제안하는 메모리 관리기는 자바 가상 머신이 기동할 때 필요한 메모리도 전부 메모리 관리기에서 할당받도록 설계하였다. 즉, 시스템 클래스 로딩, 심벌 해석, 쓰레드 등에서 동적으로 메모리 할당을 받을 필요가 있는데 이를 시스템에서 받지 않고 메모리 관리기에서 받도록 설계하였다. 물론, 자바 가상 머신 기동에 필요한 메모리를 malloc()로 받고 사용이 끝난 후 free() 로 해제 할 수도 있지만, 자바 가상 머신이 관리하는 힙에서 할당받으면 더 빠르게 할당받을 수 있을 뿐만 아니라 빈번하게 이 함수들을 호출하므로 전체적으로 메모리 할당과 해제에 드는 시간을 단축시킬 수 있다. 하지만, 힙의 영역을 일부 차지하므로 GC가 호출되는 가능성을 좀 더 높이게 된다. 그렇지만, 자바 가상 머신 내부에서 동적으로 할당 받은 객

체들은 일부 객체들을 제외하고 대부분 가상 머신이 기동하고 나서 바로 가비지가 되기 때문에 최초의 가비지 컬렉터가 수행된 이후 모두 가비지로서 회수되어 전체적으로 가비지 컬렉터에 미치는 부하는 미미하다고 할 수 있다.

3.5 논의 사항

메모리 할당과 가비지 컬렉션을 반복하다 보면 메모리 단편화가 발생한다. 단편화는 블록 내에서 사용할 수 없는 잉여 공간이 발생하는 내부 단편화와 자유 블록이지만 사용할 수 없을 만큼 작은 블록이 발생하는 외부 단편화 두가지가 있다. 자유 셀이 객체 크기보다 클 때는 메모리 블록을 분할함으로써 자유 셀이 객체 크기보다 작을 때는 메모리 블록을 병합함으로써 메모리 단편화를 줄일 수 있다. 단편화율을 줄이기 위해 표기 후 가비지를 쓸어 담는 과정에서 압축하는 방법을 생각해 볼 수 있다. 하지만 단편화율이 객체 생성이 늘어남에 따라 비례하여 증가한다면 문제가 되겠지만, 일정 비율로 유지된다면 굳이 압축을 하지 않아도 될 것이다. 본 논문에서 제안한 방법으로 객체 할당을 했을 경우 단편화율이 크게 변화하지 않음을 실험을 통해 알 수 있었다. 따라서, 본 논문에서는 프로그램 실행 시간에 부하가 될 수 있는 압축을 수행하지 않았으며, 효율적인 메모리 설계로 인해 압축에 의한 이득은 별로 크지 않음을 알 수 있었다.

4. 4-색상 기반 가비지 컬렉터 제안

가비지 컬렉션은 일반적으로 두 가지의 기본적인 일을 수행한다. 첫 번째는 객체들 중에서 가비지 객체를 식별하는 것이다. 두 번째는 가비지에 의해 사용된 힙 메모리를 회수하고 다른 프로그램이 이를 사용할 수 있도록 하는 것이다. 본 장에서는 가비지를 판단하기 위한 루트셋과 가비지 컬렉션의 기반이 되는 3-색상 기반 가비지 컬렉터에 대해 설명하고 멀티쓰레드 환경에서 동작하기 위한 개선된 4-색상 기반 가비지 컬렉터를 제안한다.

4.1 루트셋 정의

메모리 할당된 객체가 가비지인지의 판별 여부는 먼저 루트셋을 정의하고 객체들이 그 루트셋으로부터 도달 가능한지의 여부에 의해 결정된다. 본 논문에서 정의한 루트셋은 크게 세 가지로 이루어져 있다. 첫 번째는 자바 프로그램에서 공통적으로 사용하는 정보들로서 클래스 테이블, UTF8 테이블, 문자열 테이블 등이 있다.

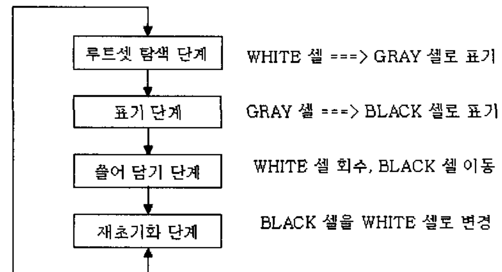
자바 가상 머신이 기동할 때 시스템 클래스들이 로딩되는데, 이 때 메모리로 로딩된 클래스들은 클래스 테이블에 저장되며 우리는 해쉬 테이블을 이용해 이를 구현하였다. UTF8 테이블은 클래스 명을 저장하는 테이블로서 해시 테이블로 구현하였다. 문자열 테이블은 함수명 등을 저장하는 문자열 상수들을 저장한다. 이 역시 해시 테이블로 구현하였다.

두 번째는 자바 스택이다. 자바 스택은 메쏘드 프레임과 각 메쏘드 프레임의 지역 변수들을 관리한다. 자바 스택은 오퍼랜드 스택으로도 활용이 된다. 본 논문에서는 쓰레기 수집기에 의한 지연 시간을 최소화하기 위해 보수적인 쓰레기 수집 방법을 채택하였다. 보수적인 쓰레기 수집기에서는 객체의 타입 정보는 보지 않고 단지, 자바 스택의 바닥부터 4byte 씩 훑어 가기 때문에 실제로 가비지인 것들도 가비지가 아닌 것으로 간주해 모든 가비지가 100% 회수 된다는 보장이 없다. 자바 스택에 저장되어 있는 객체들에 대한 포인터는 실제 객체 뿐만 아니라, 연산의 중간 결과 값도 저장되어 있어, 이 둘 간의 구별을 하기 힘들 뿐만 아니라, 모든 가비지를 100% 회수하려면, 객체의 타입 정보를 별도로 관리해야 하는데 그에 따른 부하도 상당하다.

세 번째는 네이티브 코드에서 생성한 객체들이다. 자바는 JNI를 통해 자바 프로그램에서 C 언어 혹은 다른 언어로 작성된 코드를 호출할 수있다. EVM은 네이티브 코드에서 요구하는 객체를 메모리 관리기에서 생성하도록 설계하였으며 이러한 객체들은 자바 스택에서 추적할 수 없으므로 별도로 관리가 되어야 한다.

4.2 3-색상 기반 표기-쓸어담기 가비지 컬렉터

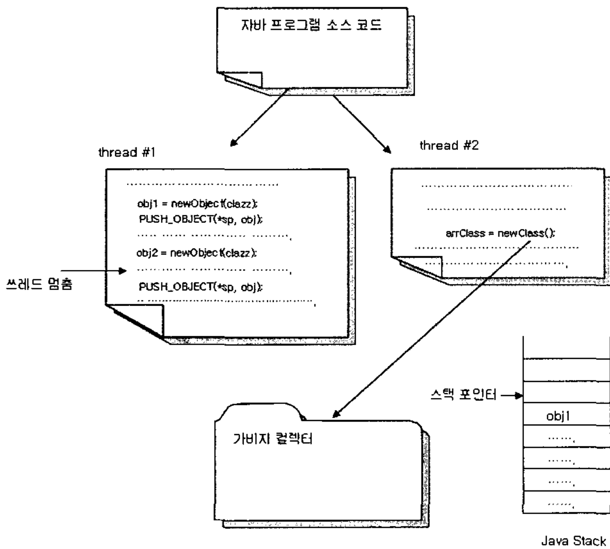
본 논문은 구현이 쉽고 다른 알고리즘에 비해 부하가 적은 3-색상 기반 표기-담기 가비지 컬렉터[15]를 기반으로 한다. 이 방식은 힙의 셀에 방문할 때 WHITE, GRAY, BLACK 중 한 가지 색을 부여한다. 가비지 컬렉션 시작 단계에서는 모든 셀은 WHITE 색상을 갖는다. 그리고, 한 번 이상 방문한 셀은 GRAY 색상을 갖게 되는데 그 셀과 그 셀의 자손 셀들까지 모두 방문했으면 BLACK 색상을 갖는다. GRAY 셀은 다시 고려를 하여 자손 셀을 방문한다. 루트셋으로부터 모든 도달 가능한 셀들이 BLACK으로 되면 표기 단계가 끝났음을 의미한다. 그리고 WHITE 로 표기된 셀은 가비지로서 회수되며, BLACK로 표기된 모든 셀은 다시 WHITE로 재초기화 됨으로써 가비지 컬렉터의 한 사이클을 마치게 된다.



(그림 3) 3-색상 기반 표기-쓸어담기 가비지 컬렉터의 동작 흐름도

4.3 3-색상 기반 표기-쓸어담기 가비지 컬렉터의 문제점

3-색상 기반 표기-담기 가비지 컬렉터는 단일 쓰레드 환경에서는 잘 동작하시만 멀티 쓰레드 환경에서는 심각한 문제를 야기할 수 있다.



(그림 4) 3-색상 표기-쓸어담기 알고리즘의 문제점

(그림 4)를 보면 자바 응용 프로그램이 2개의 쓰레드를 동시에 기동하고 있는 프로그램의 예이다. thread #1과 thread #2 의 소스 코드는 자바 가상 머신의 내부 구현 코드이다. thread #1에서 객체를 할당받은 후(newObject()) 그 객체를 obj1과 obj2에 저장하고 있다. PUSH_OBJECT() 함수를 수행하면 객체가 자바 스택(루트셋)에 들어가는데 obj1은 PUSH_OBJECT() 함수를 수행함으로써 자바 스택에 저장되었다. 그런데, obj2는 이 함수가 호출되기 이전에 thread #2가 가비지 컬렉터를 호출함으로써 thread #1이 멈추었다고 가정해 보자. 이 상황은 obj2가 메모리 관리기로부터 메모리 할당을 받았으나 자바 스택으로 넘어가지 않은 상태이며 이것은 자바 스택 메모리 구조에도 나타나 있다. 가비지 컬렉터는 obj2가 루트셋으로부터 도달가능하지 않다고 판단할 것이며, 가비지로 회수한다. 따라서, 가비지 컬렉션이 끝나고 다시 thread#1이 기동하면 obj2는 실제 메모리 주소 값을 잃어 버린 후가 되므로 자바 가상 머신은 오동작을 하는 심각한 상황에 빠진다.

4.4 해결 방법

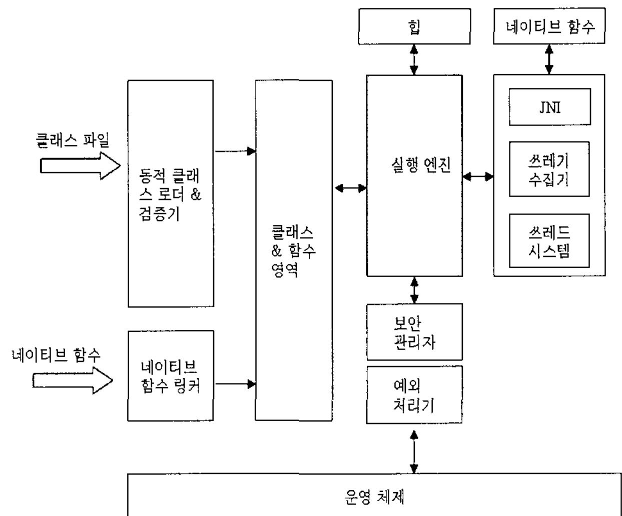
이런 문제를 해결하기 위해서 본 논문에서 4-색상 기반 표기-쓸어담기 가비지 컬렉터를 제안한다. 제안하는 방법은 3-색상 이외에 YELLOW라는 색상을 하나 더 둔다. 객체가 생성되면 모든 객체는 WHITE 색상이 아닌 YELLOW 색상을 갖는다. YELLOW 색상의 의미는 이 객체가 아직 루트셋으로 저장 되기 이전 상태라는 의미이며 가비지 컬렉터시 이들을 회수의 대상에서 제외시키라는 의미이다. 그리고 YELLOW 색상은 적절한 시점에 WHITE 색상으로 변경해줘야 한다. 따라서, 가비지로 간주된 셀 중에 WHITE 색상은 메모리에서 해제해도 되지만, YELLOW로 표기된 셀은 위에서처럼 루트셋에 할당이 안 된 살아 있는 객체이므로 메모리 해제에서 제외시켜야 한다. 본 논문에서는 YELLOW 색상 객체를 관리하는 메모리를 따로 두어 관리하도록 하였다. 객체가 힙으로부터 메모리를 할당 받으면 1차적으로 YELLOW 색상을 갖도록 하였고, 경우에 따라 적절한 시점에 YELLOW를 WHITE로 변경해 주는 함수를 호출하도록 하였다. 이렇게 함으로써 멀티쓰레드에서 프로그램이 멈춰섰던 문제점을 말끔히 해결 할 수 있었다. YELLOW 색상을 갖는 객체는 매우 짧은 시간동안 YELLOW 색상을 갖다가 곧, WHITE 색상으로 변색되기 때문에, YELLOW 색상을 갖는 객체는 전체 객체 중 일부분이므로 YELLOW 색상을 위한 메모리를 별도로 관리하더라도 이에 대한 부하는 크지 않다. 또한 효율적인 메모리 관리 설계로 인해, 해당하는 객체에 접근하여 색상을 변경하는 데 드는 시간은 미미해서 쓰레기 수집기 기동 시간에 별 영향을 미치지 않았다.

5. 실 험

본 장에서는 임베디드용 자바 가상 머신을 소개하고 이 자바 가상 머신에서 구현된 쓰레기 수집기의 성능을 정량적으로 보인다. 먼저 성능 평가를 위한 실험 환경을 설명하고, 다양한 실험을 통하여 가비지 컬렉터의 성능 변화를 관찰한 후 이를 분석한다.

5.1 EVM 소개

(그림 5)는 한국전자통신연구원에서 개발한 EVM 자바 플랫폼의 전체 구조를 보여준다. 자바플랫폼의 구현은 크게 가상 머신과 자바 API로 구분할 수 있다. 본 연구에서는 CDC 플랫폼을 위한 표준 규격의 자바 가상 머신을 네이티브 메소드 호출 인터페이스를 포함하여 클린룸으로 구현하였다. 자바 API를 위해서는 오픈소스 프로젝트인 Classpath [16] 버전 0.05를 기반으로 PP(Personal Profile) 규격의 클래스 라이브러리를 개발하였으며 자바 가상 머신과 통합하여 CDC/PP 규격의 자바플랫폼을 완성하였다. 개발 환경은 리눅스 상에서 GTK+1.2 그래픽 패키지를 기반으로 한다.



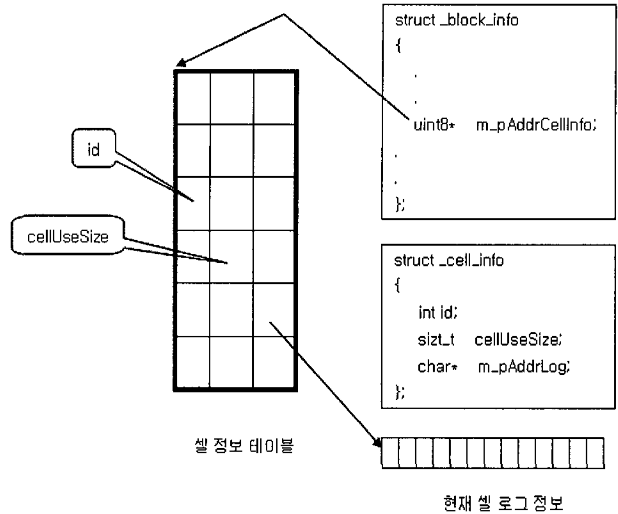
(그림 5) EVM의 자바 플랫폼 구조

EVM은 CDC 플랫폼을 포함하여 J2SE 및 J2EE에도 사용 가능한 완전한 Java2 가상 머신 이다. 자바 가상 머신의 기본 동작은 로컬 파일이나 네트워크으로부터 읽어 들인 클래스파일을 해석하여 클래스 정보를 메소드 영역과 힙 등에 저장하고 저장된 정보를 기반으로 순서에 따라 일련의 메소드들을 수행시키는 것이다. 본 구현에서는 JNI(Java Native Interface)와 동적 라이브러리 링킹을 지원하므로 프로그래머가 네이티브 메소드를 자유롭게 정의하여 사용할 수 있으며 코드 검증기 구현을 통해 안정성 및 보안 기능을 강화하였다. 특히 자원 제약이 많은 임베디드 시스템 환경에서 우수한 성능을 보장하도록 쓰레드 시스템, 자바 메모리 시스템 및 쓰레기 수집기 등을 설계 및 구현하였다.

5.2 메모리 관리기의 정보 출력 API

메모리 사용 상태를 측정하거나 감시하기 위해 메모리 프로파일 정보를 출력하는 API를 구현하였다. 메모리 프로파일 정보를 이용해 메모리의 사용 정보와 통계 정보를 얻을 수 있어서 효율적인 메모리 관리기를 설계하는데 도움을 받을 수 있었다. 메모리 관리기를 디버깅하거나 프로파일 정보를 얻기 위해 블록 정보 구조체에 셀 정보를 저장하고 있는 새로운 구조체를 추가하였다. (그림 6)은 기존의 블록 정보 구조체에 셀 정보를 위한 구조체가 새로 추가된 모습을 보여준다.

(그림 6)의 구조체에서 id는 셀에 부여 되는 식별자이다. cellUseSize는 셀에 할당된 메모리에서 실제 사용하는 메모리 양을 나타낸다. 실제 사용하는 메모리의 양은 내부 메모리 단편화를 측정 하는데 사용된다. m_pAddrLog은 셀에 대한 로그 내용을 나타내는 메모리를 지정 한다. 이 정보를 바탕으로 <표 2>와 같이 메모리 프로파일 정보를 출력하는 API를 구현하였다.



(그림 6) 셀 정보 구조체

〈표 2〉 메모리 프로파일 정보 출력 API

함수명	기능
mm_print_memory_usage_area ()	힙 사용 전체 영역을 출력.
mm_print_memory_cell_usage_info ()	셀 타입에 따른 사용 통계 정보 출력
mm_print_memory_state ()	블록 상태에 따른 힙 사용 정보 출력
mm_print_memory_use_fill_empty_block_info ()	Use, Filled(Large), Empty Block 정보 출력
mm_print_blockinfo()	Block_Info를 이용해 해당 블록 정보 출력
mm_print_blockinfo_for_id ()	id를 이용해 해당 블록 정보 출력
mm_print_blockinfo_for_object ()	객체 주소를 이용해 해당 블록 정보 출력
mm_print_blockinfo_for_cell ()	셀 주소를 이용해 해당 블록 정보 출력
mm_print_cell_color_for_addr ()	셀 주소를 이용해 셀 색상 출력
mm_print_cell_type_for_addr ()	셀 주소를 이용해 셀 타입 출력
mm_print_cell_size_for_addr ()	셀 주소를 이용해 셀 크기 출력
mm_print_block_id ()	GC invoke와 메모리 할당으로 인한 블록 변화를 점검 하기 위해 Small, Filled(Large), Empty Block 정보 출력
mm_print_external_fragmentation ()	블록의 외부 단편화 정보 출력
mm_print_internal_fragmentation ()	셀의 내부 단편화 정보 출력
mm_print_block_external_fragmentation ()	특정 블록의 외부 단편화 정보 출력
mm_print_block_internal_fragmentation ()	특정 블럭에 대한 셀의 내부 단편화 정보 출력

5.3 실험 환경 구축

자바 가상 머신은 기동 시 시스템 클래스 로딩 등 상당량의 메모리를 소비한다. 따라서 적은 메모리를 요구하는 자바 응용 프로그램에서도 결과적으로 많은 양의 메모리를 요구하는 것처럼 보이기 때문에 그대로 실험하면 실험 데이터에 왜곡이 생길 수 있다. 또한, 메모리 관리기나 가비지 컬렉터 동작을 디버깅 하는데도 장애 요소가 된다. 따라서, 본 논문에서는 실험을 위해 실험용 자바 응용 프로그램이 필요

〈표 3〉 EVM 과 Bare EVM 의 클래스 로딩 차이

	EVM	Bare EVM
Thread	Multi Thread 사용	단일 Thread 사용
java/lang/Object	로딩	로딩
java/lang/Clone	로딩	로딩
java/lang/Class	로딩	로딩
java/lang/String	로딩	로딩
java/lang/Thread	로딩	비 로딩
java/lang/ThreadGroup	로딩	비 로딩
java/lang/Throwable	로딩	비 로딩
java/io/Serializable	로딩	비 로딩
java/lang/System	로딩	비 로딩
java/lang/Void	로딩	비 로딩
java/lang/Boolean	로딩	비 로딩
java/lang/Byte	로딩	비 로딩
java/lang/Character	로딩	비 로딩
java/lang/Short	로딩	비 로딩
java/lang/Integer	로딩	비 로딩
java/lang/Long	로딩	비 로딩
java/lang/Float	로딩	비 로딩
java/lang/Double	로딩	비 로딩
java/lang/Throwable	로딩	비 로딩
java/lang/Exception	로딩	비 로딩

로 하지 않는 시스템 클래스들 가상 머신 기동시 로딩되지 않도록 하였다(이하 Bare EVM 이라 칭함). 이렇게 함으로써 메모리 구조를 단순화 시킬 수 있었고 메모리 관리기와 가비지 컬렉터의 디버깅을 원활히 할 수 있었다. <표 3>은 Bare EVM에서 로딩하는 클래스들을 보여주고 있다.

<표 4>는 가상 머신 기동 시 초기 메모리 사용량에 대한 비교 결과이다. 표에서 볼 수 있듯이 Bare EVM은 EVM의 47%의 만의 메모리만 사용하고 있음을 알 수 있다.

외부 메모리 단편화율은 자유 블록이지만 객체가 할당될 수 없을 만큼 작아서 낭비되는 메모리가 얼마나 되는지 나타내는 지표로서 블록의 개수를 n, 블록의 셀 크기를 Cs, 셀의 총 개수를 Ct, 사용되지 않은 셀의 개수를 Cf 이라고 하면 식 (1), (2), (3)과 같이 계산될 수 있다.

$$allocMemory = \sum^{n} Cs \times Ct \quad (1)$$

$$useMemory = \sum^{n} Cs \times (Ct - Cf) \quad (2)$$

$$Fe = (allocMemory - useMemory) \div (allocMemory) \times 100 \quad (3)$$

식 (1)에서 *allocMemory*는 모든 블록에 할당된 메모리 전체 크기이고 식 (2)에서 *useMemory*는 블록중 사용된 셀들의 메모리 크기 합이고 식 (3)에서 Fe 는 외부 단편화율이다.

내부 단편화율은 사용된 셀 중에서 사용되지 않은 메모리가 얼마나 되는지 나타내는 지표로서 셀의 총 개수를 n, 할당된 셀의 크기를 Cs, 사용된 셀의 실제 셀 크기를 Cu 라고 하면 식 (4), (5), (6)과 같이 계산될 수 있다. .

$$allocMemory = \sum^{n} Cs \quad (4)$$

$$useMemory = \sum^{n} Cu \quad (5)$$

$$Fi = (allocMemory - useMemory) \div (allocMemory) \times 100 \quad (6)$$

식 (4)에서 *allocMemory*는 모든 셀의 메모리 전체 크기이고 식 (5)에서 *useMemory*는 *셀중* 사용된 셀들의 메모리 크기 합이고 식 (6)에서 Fi는 내부 단편화율이다.

〈표 4〉 가상 머신 구동시 사용하는 메모리 비교

	외부 메모리 단편화			내부 메모리 단편화		
	전체할당 메모리 (byte)	사용 메모리 (byte)	단편화율 (%)	전체할당 메모리 (byte)	사용 메모리 (byte)	단편화율 (%)
EVM	688,940	615,756	10	615,756	518,853	15
Bare EVM	324,028	237,564	26	237,564	209,796	11

5.4 객체 생성 실험 결과

임의의 크기를 갖는 1000 개의 객체를 생성하는 실험을 하였다. <표 5>은 객체 생성 전과 후의 전의 메모리 정보이다.

〈표 5〉 1000개의 객체 생성 실험 결과

		블록수 (개)	메모리 크기 (byte)	점유율 (%)	내부 단편화율 (%)	외부 단편화율 (%)
객체 생성 전	할당된 전체 블록	250	2,744,320	100	9	2
	조금 채워진 블럭	16	131,072	4		
	전부 채워진 블럭	234	2,613,248	95		
	미사용 블럭	0	0	0		
객체 생성 후	할당된 전체 블럭	253	2,768,896	100	9	2
	조금 채워진 블럭	16	131,072	4		
	전부 채워진 블럭	237	2,637,824	95		
	미사용 블럭	0	0	0		

〈표 6〉 Webjabi 실험 결과

		블록수 (개)	메모리 크기 (byte)	점유율 (%)	내부 단편화율 (%)	외부 단편화율 (%)
웹페이지 로딩 전	할당된 전체 블록	395	4,145,152	100	10	1
	조금 채워진 블록	18	147,456	3		
	전부 채워진 블록	377	3,997,696	96		
	미사용 블록	0	0	0		
웹페이지 로딩 후	할당된 전체 블록	2,147	22,405,120	100	10	0
	조금 채워진 블록	18	147,456	0		
	전부 채워진 블록	2,129	22,257,664	99		
	미사용 블록	0	0	0		

객체를 생성하기 전의 메모리 사용은 2,744,320 bytes이며, 객체를 생성한 이후의 메모리 사용은 2,768,896 bytes이므로, 1000 개의 객체를 생성하게 되면 약 24K 가량의 메모리가 소요 됨을 알 수 있다. 객체 생성 전과 후의 단편화율은 큰 변화가 없었다.

두 번째는 실제로 사용되고 있는 자바 응용 프로그램으로 실험을 하였다. 소규모가 아닌 대용량의 객체를 생성할 수 있는 Webjabi라는 프로그램으로 실험하였다. Webjabi 는 한국전자통신연구원에서 만든 임베디드용 자바 웹 브라우저이다. <표 6>은 이에 대한 실험 결과이다.

본 실험을 통해 대용량의 객체도 충분히 생성 및 관리할 수 있음을 확인할 수 있었다. 또한 단편화율도 우려했던 것보단 크지 않음을(10% 이내)을 확인할 수 있었다.

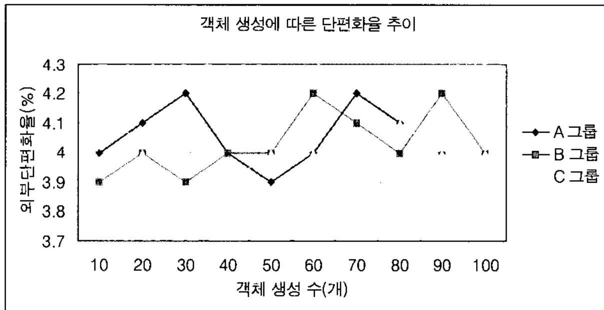
본 논문에서 제안하는 메모리 관리기가 생성하는 객체 수가 많아짐에 따라 단편화율이 어떻게 변화하는지 추이를 보기 위한 실험을 수행하였다. 메모리 프로파일 정보를 보는 기능을 자바 프로그램에서 제어할 수 있도록 메모리 프로파일 클래스 구현이 필요하다. 본 논문에서는 디버깅 및 프로파일용으로 개발한 C API를 JNI(Java Native Interface)를 통해 자바에서도 호출할 수 있도록 <표 8>와 같은 API를 구현하여 이를 실험에 이용하였다.

이 클래스의 API를 이용하면 자바 프로그램에서 객체를 생성한 후 원하는 시점에서 메모리 프로파일 정보를 보는 것이 가능해진다.

실험을 위해 생성하는 객체 집단을 세 부류로 분류하였다. A 그룹은 크기가 10인 int형 배열을 계속해서 생성하는

〈표 8〉 메모리 프로파일 출력 용 자바 API

함수명	기능
printMemoryUsageArea()	힙 전체 영역 출력
printMemoryCellUsageInfo()	셀 타입에 따른 사용 통계 정보 출력
printMemoryState()	블록 상태에 따른 힙 사용 정보 출력
printMemoryUseFillEmptyBlockInfo()	Use, Filled(Large), Empty Block 정보 출력
printMemoryBlockInfoForObject()	객체 주소를 이용해 해당 블록 정보 출력
printMemoryBlockId()	쓰레기 수집기 기동과 메모리 할당으로 인한 블록 변화를 점검하기 위해 조금 채워진 블록, 전부 채워진 블록, 미사용 블록 정보 출력
printMemoryExternalFragmentation()	블록의 외부 단편화 정보 출력
printMemoryInternalFragmentation()	셀의 내부 단편화 정보 출력
printGcAnchoredObject()	객체 내용 출력
printGcCellColor()	색상 정보 출력
printMemoryRootSetInfo()	루트셋에 대한 정보 출력



(그림 7) 객체 생성에 따른 단편화율 추이

그룹이고, B 그룹은 크기가 10인 double 형 배열을 계속해서 생성하는 집단이고 C 그룹은 크기가 10인 int형 배열과, double형 배열, float형 배열을 한꺼번에 계속해서 생성하는 그룹이다. (그림 7)은 이 실험의 결과를 보여주고 있다.

(그림 7)에서 볼 수 있듯이, 객체 생성 수가 많아지더라도 외부 메모리 단편화는 거의 일정함(4%)을 알 수 있다.

6. 결 론

본 논문에서는 임베디드 자바 가상 머신인 EVM에서의 메모리 관리기 및 가비지 컬렉터를 제안하였다. 본 논문에서 제안하는 메모리 관리기는 자유-리스트 방식을 채택하였다. 이 방식은 단편화가 발생하는 단점이 있지만, 크기별로 분할된 자유 리스트에서 필요한 크기에 맞는 메모리 공간을 얻어 오기 때문에 메모리 할당이 빠르고 분할된 셀들이 링크드 리스트로 연결되어 있어 메모리 해제를 빠르게 할 수 있다. 또한 메모리 크기를 작은 크기로 세분화 하여 링크드 리스트로 관리하기 때문에 작은 객체들을 쉽게 관리할 수 있고 메모리 확장이 용이한 장점이 있다.

가비지 컬렉터로는 3-색상 기반 표기-쓸어담기 가비지 컬렉터를 기반으로 채택하되 멀티쓰레드 지원이 가능하도록 개선된 4-색상 기반 가비지 컬렉터를 제안하였다. 기존의 WHITE/GRAY/BLACK 이외에 YELLOW 라는 색상을 추가로 도입하여 멀티쓰레드 환경에서 가상 머신이 멈춰서는 문제점을 해결할 수 있었다.

본 논문에서 구현한 메모리 관리기 및 가비지 컬렉터는 EVM에 탑재되어 잘 동작하고 있음을 확인할 수 있었다. 또한 실험을 통해 메모리 관리기의 성능을 보였으며 단편화율이 크게 문제가 되지 않음을 보였다.

향후에는 제안하는 기법을 기반으로 단편화율을 줄일 수 있는 방법에 대한 연구와 객체의 타입 정보를 이용한 모든 가비지 회수 방법에 대한 연구를 수행할 예정이다.

참 고 문 헌

[1] Sun Microsystems, 'Java2 Platform, Micro Edition, Connected Device Configuration(CDC)', http://java.sun.com/products/cdc/index.jsp, 2005.

[2] T. Lindholm, F. Yellin, 'The JavaTM Virtual Machine Specification', 2nd Ed., Addison-Wesley, 1999.

[3] D. A. Barrett and B. G. Zorn, "Using lifetime predictors to improve memory allocation performance," In Proceedings of SIGPLAN Conference on Programming Languages Design and Implementation(PLDI), Vol.24, No.7, pp.187-196, June, 1993.

[4] S. M. Blackburn, P. Cheng, and K. S. McKinley, "Myths and reality: The performance impact of garbage collection," In Proceedings of International Conference on Measurement and Modeling of Computer Systems, pp.25-36, June, 2004.

[5] H. Lieberman and C. E. Hewitt, "A real-time garbage collector based on the lifetimes of objects," Communications of the ACM, Vol.26, No.6, pp.419-429, 1983.

[6] W. Liu, Z. Chen, and S. Tu, "Research and analysis of garbage collection mechanism for real-time embedded java," In Proceedings of International Conference on Computer Supported Cooperative Work in Design, pp.462-468, May, 2004.

[7] www.kaffe.org

[8] Chia-Tien Dan Lo, Witawas Srisa-an and J. Morris Chang, "Who is collecting your java garbage?," IT Pro. IEEE Computer Society, pp.44-50, March April, 2003

[9] D. Doligez, and X. Leroy, "A Concurrent, Generational Garbage Collector for a Multithreaded Implementation of ML," In Proc. 20th ACM Symp. Principles of Programming Languages, ACM Press, pp.113-123, 1993.

[10] http://java.sun.com

[11] George E. Collins, "A method for overlapping and erasure of lists," Communications of the ACM, Vol.3, No.12, pp.655-657, December, 1960.

[12] John McCarthy, "Recursive functions of symbolic expressions

and their computations by machine," Communications of the ACM, pp.184-195, 1960.

[13] Marvin L. Minsky, "A Lisp garbage collector algorithm using serial secondary storage," Technical Report Memo 58, Project MAC. MIT, Cambridge, December, 1963.

[14] Andrew W. Appel, "Simple generational garbage collection and fast allocation," Software Practice and Experience, Vol.19, No.2, pp.171-183, 1989.

[15] E. W. Dijkstra, L. Lamport, A. J. Martin, C. S. Scholten, and E. F. M. Steffens, " On-the-fly garbage collection: An exercise in cooperation," Communication of the ACM, Vol.21, No.11, pp.965-975, November, 1978.

[16] http://www.gnu.org/software/classpath/

이 상 윤

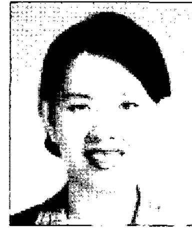
e-mail : syllee@etri.re.kr

1994년 한양대학교 전자통신공학과(학사)

1996년 한양대학교 전자통신공학과(석사)

1999년 9월~현재 한국전자통신연구원 임베디드S/W연구단 선임연구원

관심분야 : 무선인터넷플랫폼, JVM, 임베디드 시스템



원 희 선

e-mail : hswon@etri.re.kr

1990년 2월 연세대학교 전산과학(학사)

1992년 2월 한국과학기술원 전자계산학(석사)

2000년 5월 현제 한국전자통신연구원 임베디드S/W연구단 선임연구원

관심분야 : JVM, 임베디드 시스템, 데이터베이스

최 병 욱

e-mail : buchoi@hanyang.ac.kr

1973년 한양대학교 전자공학과(학사)

1978년 일본 경응의숙(KEIO) 대학 전기공학과(석사)

1981년 일본 경응의숙(KEIO) 대학 전기공학과(박사)

현 재 한양대학교 정보통신대학 정보통신학부 교수

관심분야 : 영상처리, 멀티미디어 공학