메모리가 적은 자바 시스템을 위한 자동 동적 메모리 관리 기법

(Automatic Dynamic Memory Management Techniques for Memory Scarce Java system)

최 형 규 † 문 수 묵 ††

(Hyung-Kyu Choi) (Soo-Mook Moon)

요 약 많은 내장형 시스템들이 자바(Java)를 널리 채택하고 있다. 내장형 시스템은 자바 가상 머신을 통해 자바를 지원하며, 자바 가상 머신은 쓰레기 수집기(Garbage Collector)를 통해서 동적 메모리를 자동으로 관리한다. 내장형 시스템은 적은 메모리를 가지고 있기 때문에 자바 가상 머신은 이를 효율적으로 관리해야 한다. 본 논문에서는 여러 자바 프로그램을 동시에 실행할 수 있는 자바 가상 머신에서 적은 메모리를 사용하면서도 효과적으로 메모리를 관리할 수 있는 메모리 관리 기법을 제안한다. 우선 개선된 압축(compaction)기법 기반의 쓰레기 수집 기법을 소개하여 움직일 수 없는 메모리 영역이 존재하더라도 외부 단편화(external fragmentation) 문제를 극복한다. 다음으로 수행 중 메모리 사용을 줄이기 위해서 쓰레기 수집기가 메모리에서 필요 없는 클래스(class)들을 선택적으로 수거하는 class unloading 기법을 소개한다. 소개한 기법들을 실제 동작하는 내장형 시스템에서 실험한 결과, 메모리가 부족하여 동시에 수행할 수 없었던 프로그램들이 같이 수행되는 등 매우 효과적이었다.

키워드 : 자바 가상 머신, 쓰레기 수집기, 압축 기법, 클래스 언로딩, 메모리 단편화

Abstract Many embedded systems are supporting Java as their software platform via Java virtual machine. Java virtual machine manages memory automatically by providing automatic memory management, i.e. garbage collector. Because only scarce memory is available to embedded system, Java virtual machine should use small memory and manage it efficiently. This paper introduces two memory management techniques to exploit small memory in Java virtual machine which can execute multiple Java applications concurrently. First, compaction based garbage collection is introduced to overcome external fragmentation problem in presence of immovable memory area. Then garbage collector driven class unloading is introduced to reduce memory use of unnecessary loaded classes. We implemented these techniques in working embedded system and observed that they are very efficient, since more Java applications are able to be executed concurrently and memory use is also reduced with these techniques.

Key words : Java Virtual Machine, Garbage Collector, Compaction, Class Unloading, Memory Fragmentation

· 본 연구는 2006년 정부(교육인적자원부)의 재원으로 한국학술진흥재단 (KRF-2006-311-D00757)의 연구비 지원으로 수행하였습니다.
· 이 논문은 2007 한국컴퓨터종합학술대회에서 '메모리가 적은 자바 시스템을 위한 자동 동적 메모리 관리 기법'의 제목으로 발표된 논문을 확장한 것임

† 학생회원 : 서울대학교 전기컴퓨터공학부
hectoct@altair.snu.ac.kr
†† 종신회원 : 서울대학교 전기컴퓨터공학부 교수
smoon@altair.snu.ac.kr

논문접수 : 2007년 12월 6일
심사완료 : 2008년 5월 21일

정보과학회논문지: 시스템 및 이론 제35권 제8호(2008.8)

1. 서 론

자바(Java)는 인터넷과 내장형 기기의 보급과 더불어 그리고 자바의 장점 중 하나인 플랫폼 독립성 때문에 디지털TV, 핸드폰 그리고 다양한 휴대용 이동 기기와 같은 내장형 기기에서 활용이 늘어가고 있다. 자바의 플랫폼 독립성은 자바 가상 머신(Java Virtual Machine, JVM)을 통해서 얻어지며, 내장형 기기는 자바 가상 머신을 채택하여 자바 프로그램을 수행시킬 수 있게 된다. 자바 프로그램은 바이트코드(bytecode)라는 중간 코드로 컴파일 되어 있으며 자바 가상 머신이 이를 수행하여 서로 다른 플랫폼에 같은 수행 결과가 나오도록 해주기 때문이다.

내장형 시스템은 대부분의 경우 메모리가 적기 때문에, 메모리를 효율적으로 관리해야 하며 메모리를 적게 사용해야 한다. 자바 가상 머신은 메모리를 자동으로 관리하며 이는 쓰레기 수집기(Garbage Collector)에 의해서 이루어진다. 쓰레기 수집 기법에는 다양한 방법들이 소개되었다[1]. 본 논문에서는 여러 자바 프로그램을 동시에 수행할 수 있는 내장형 시스템에서 효과적인 메모리 관리 기법을 소개하려고 한다. 우선 개선된 압축 방식의 쓰레기 수집기를 제안하여 메모리 외부 단편화(external fragmentation) 문제를 극복하여 큰 메모리 객체를 쉽게 할당할 수 있도록 한다. 다음으로 쓰레기 수집기가 사용하지 않는 클래스(class)들을 수거하여 자바 프로그램 수행 중 메모리 사용량을 줄일 수 있는 class unloading 기법을 소개한다.

다음 2장에서는 여러 자바 프로그램을 동시에 수행시킬 수 있는 자바 시스템을 소개하며 문제점을 제기한다. 3장에서는 개선된 압축 기반의 쓰레기 수집 기법을 제안하며 4장에서는 쓰레기 수집기를 이용한 class unloading 기법을 소개한다. 5장에서는 제안된 두 기법들을 어떻게 실제 내장형 시스템에 구현하였는가를 설명한다. 6장에서는 실험으로 통해 소개된 기법들의 성능을 평가하고, 마지막으로 7장에서 요약한다.

2. 여러 프로그램들을 동시 수행하는 자바 시스템

널리 사용되는 내장형 자바 시스템인 핸드폰은 다양한 멀티미디어 프로그램들을 제공하고 있다. 자바 시스템은 프로그램의 호환성과 유지 보수 면에서 많은 이점을 가지고 있으며, 핸드폰의 특성상 상호 연동하는 여러 자바 프로그램을 동시에 수행할 필요성이 제기되었다. 여러 프로그램을 동작시키는 방법은 크게 두 가지가 있다. 우선 자바 가상 머신을 여러 개 수행하는 방법이 있지만 이는 메모리와 같은 자원을 많이 요구하며 상호작용하는 기능을 구현하기 쉽지 않다. 그래서 내장형 시스템에서는 하나의 자바 가상 머신에서 여러 자바 프로그램들을 수행하는 방법을 선택하고 있다. 이와 같은 방법은 자바를 발표한 Sun Microsystems사에서도 채택한 방법으로 많은 연구가 진행되고 있다[2].

본 논문에서 사용한 자바 시스템은 기존의 SUN Microsystems사에서 발표한 내장형 시스템 자바 표준인 J2ME(Java 2 Micro-edition) CLDC(Connected Limited Device Configuration)[3]에 기반하고 있다. J2ME CLDC에서 자바 가상 머신은 KVM(K Virtual Machine)이 있으며 이는 표준 자바보다 간단한 명세를 가지고 있으며 여러 자바 프로그램을 수행할 수 있도록 되어있다[3,4].

KVM의 메모리 시스템은 기본적으로 자바 Heap과 자바 가상 머신을 위한 C Heap을 따로 관리하고 있으며 한번 늘어난 C Heap영역은 자바 Heap으로 사용하지 못하도록 되어 있다. 자바 Heap 영역에서의 메모리 조각화(memory fragmentation) 문제는 3장에서 소개하는 압축 기반의 쓰레기 수집기(compacting garbage collector)를 사용하여 극복하고 있다. 이러한 환경에서 C Heap이 늘어날 경우 자바 Heap으로 사용할 수 있는 영역이 줄어들게 되어 프로그램을 위한 메모리가 부족하게 된다. 여러 자바 프로그램이 하나의 KVM에서 동작하는 경우 이러한 문제가 더욱 자주 발생한다. 그 결과 여러 자바 프로그램을 수행하기 위해 자바 Heap과 C Heap이 메모리를 공유하도록 수정되었다. 하지만 기존의 압축 기반 쓰레기 수집기는 적용할 수 없게 되어 3장 에서 소개하는 개선된 압축 기법을 필요로 하게 되었다. 또한, 메모리 사용량을 줄이기 위해 사용하지 않는 클래스 들을 메모리에서 제거하는 class unloading 기법을 4장에서 제안한다.

3. 압축(Compaction) 기반의 쓰레기 수집 기법

압축 쓰레기 수집기는 메모리 외부 단편화(external fragmentation) 문제를 해결하는 대표적인 쓰레기 수집 기법이다[1]. 압축 쓰레기 수집기는 다음 그림 1과 같이 사용중인 메모리 영역을 한 쪽으로 모아 사용되지 않는 빈 영역을 하나로 합쳐지도록 한다.

다양한 압축 기법들이 제안되었으며[1, 5-10], 메모리가 적으면서 CPU의 속도도 느린 환경에 적합한 방법으로 Table-based compaction 기법[5-7]이 있다. 이 기

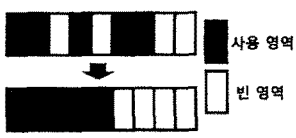

그림 1 압축 기법 적용 전과 후

법은 추가적인 메모리를 요구하지 않으며 메모리 객체의 숫자에 비례하는 복잡도를 가져 빠른 동작을 보장한다고 알려져 있다[6,10]. 본 논문의 쓰레기 수집기는 이 압축 기법에 기반하고 있다.

3.1 기존 Table-based compaction 기법

Table-based compaction은 다음과 같이 세 단계로 수행된다[9].

1. 압축 단계: 사용 중인 메모리 영역을 한 쪽으로 옮기면서 옮겨진 메모리 공간에 대한 정보를 새로이 생성된 빈 영역에 break table(BT)라는 자료구조로 만들어 저장한다. 이 과정은 다음 두 단계로 나뉜다.
   A. i번째 항목인 (a_i, Sum_i)을 계산하여 BT 생성
      a_i : 다음 빈 영역의 시작 주소
      S_i : 다음 빈 영역의 크기
      $Sum_i = Sum_{i-1} + S_i$, $i \geq 1$, $Sum_0 = 0$
   B. i번째 빈 영역 뒤에 있는 사용 중인 메모리들을 a_i 쪽으로 옮긴다. 옮기다 보면 BT가 저장되어 있는 공간에 옮겨지는 메모리가 쓰여지기 때문에 BT를 다음 빈 영역으로 대피시키는 "rolling"을 수행한다. 아직 옮겨지지 않은 사용 중인 영역이 있으면 A 단계부터 반복한다.
2. BT 정렬 단계: 다음 단계의 속도를 높이기 위해 BT를 시작 주소인 a_i의 크기 순으로 정렬한다.
3. 참조 포인터 갱신 단계: 옮겨진 메모리 영역을 참조하고 있던 포인터들의 주소 값을 새로운 주소 값으로 갱신한다. 원래 주소가 p라고 할 때, $a \leq p < a'$ 인 (a, S)와 (a', S')를 BT에서 찾아 포인터의 값을 p-S로 바꿔 주면 된다.

그림 2는 빈 영역이 두 개로 나뉘어져 있는 메모리를 압축한 후 2개의 (a, S)항목이 BT에 생성되는 과정을 보여주고 있다. 이러한 과정을 수행하다 보면 최악의 경우(worst-case) 빈 영역의 개수에 비례하는 크기의 BT가 생성된다. 하지만 빈 영역의 최소 크기가 하나의 항목을 가지고 있는 BT의 크기와 같다면 빈 공간에 필요한 정보를 모두 저장할 수 있다고 알려져 있다[5].

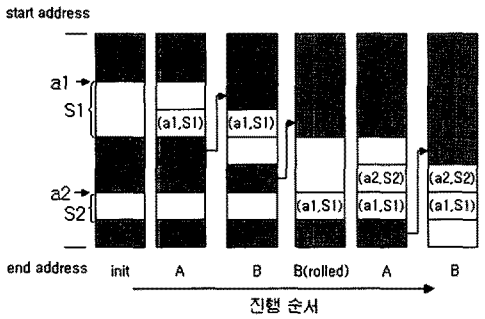

그림 2 Table-based compaction 예제

3.2 개선된 Table-based compaction 기법

앞의 Table-based compaction은 모든 메모리 영역이 움직일 수 있어야 적용 가능하다. 하지만 소개된 자바 시스템은 자바 객체와 C 객체를 같은 메모리에 할당하며 C언어의 malloc 등으로 할당한 메모리 객체는 움직일 수 없다. 이 경우 C 객체를 옮기면 안 되므로 앞에서 소개한 압축 기법을 그대로 적용할 수 없게 된다. 본 논문에서는 이러한 C 객체들은 pinning[9]이라는 방법으로 움직이지 못하게 한 후 다음과 같은 방식으로 압축을 수행하였다. 일반적인 경우 다음 그림 3처럼 움직일 수 있는 객체와 움직일 수 없는 객체가 메모리에 같이 존재할 것이다.

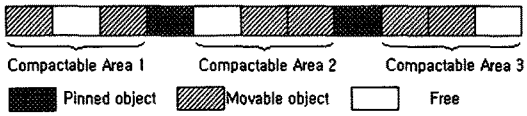

그림 3

전체 메모리를 pinned object를 기준으로 움직일 수 있는 객체와 빈 영역으로 이루어진 compactable area로 구분하자. 각 compactable area에 앞에서 설명한 압축 기법의 1단계를 적용하면 다음 그림 4와 같이 BT가 나뉘어 존재하게 된다. m개로 나눠진 BT의 위치를 알기 위해서는 m에 비례하는 크기의 메모리 공간이 필요하다. 각 BT을 linked list와 같은 형태로 만들게 되면 크기가 증가되어 빈 영역에 BT가 저장될 수 있다고 알려진 특성이 깨어지므로 BT자료구조에 정보를 추가할 수는 없다. 또한 쓰레기 수집기가 동작 중이므로 새로운 메모리를 할당 받을 수도 없다.

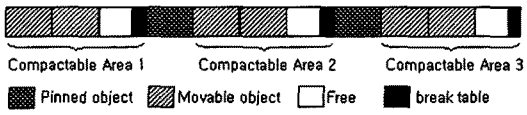

그림 4

본 논문에서는 BT들의 위치를 C stack에 저장하여 이 문제를 해결하였다. 하지만 BT의 개수 m에 매우 커지면 이에 비례하는 크기의 매우 큰 C stack공간이 필요하게 되어 문제가 된다. 본 논문에서는 최대 M개의 BT이 위치를 저장하는 고정된 크기의 메모리를 C stack에서 할당 받아 최대 M개의 compaction area에 대하여 압축 1단계를 적용한 후 2단계와 3단계를 수행하도록 하였다. 그리고 compaction area가 M보다 많다면 2단계와 3단계가 여러 번 수행된다. 이와 같은 방법으로 움직일 수 없는 메모리가 존재하는 경우에도 효율

적이고 검증 받은 Table-based compaction 기법을 적용 할 수 있다.

4. Class Unloading

Class unloading은 자바 가상 머신 명세에서 소개되는 것으로 사용하지 않는 클래스들을 메모리에서 제거하여 메모리 사용량을 줄이는 방법이다[11]. 이를 위해서는 사용하지 않는 클래스들을 찾아야 하는데 추가적인 자료구조 없이 빠르게 찾기 어려워 내장형 자바 시스템에는 일반적으로 적용하지 않는다. 본 논문에서 소개한 자바 시스템은 여러 프로그램을 동시에 수행시키기 때문에 종료된 자바 프로그램에서 사용하던 클래스들을 메모리에서 제거하여 다른 프로그램에서 사용할 메모리 공간을 확보해야 한다. 여기서는 쓰레기 수집기가 사용하지 않는 클래스들을 쉽고 빠르게 찾아 메모리에서 제거하는 방법을 제안한다.

4.1 Two-level Class Table

자바 가상 머신은 일반적으로 다음 그림 5와 같은 hash table기반의 하나의 class table을 사용하며, 클래스의 사용 여부를 판단하기 위해서는 모든 객체들을 방문하여 해당 클래스를 사용하는지 확인해야 하며 이는 시간이 많이 걸리게 된다.

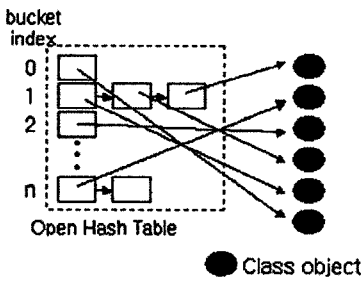

그림 5 Single Class Table

본 논문에서는 그림 6과 같이 두 종류의 class table을 도입하였다. Global class table은 system 클래스에 대한 정보를 가지고 있으며 local class table은 자바 프로그램을 위한 클래스들을 관리한다. 이를 활용하면 프로그램이 종료될 때 해당 프로그램을 위한 local class table에 속해 있던 클래스들은 사용되지 않는다는 것을 쉽게 알 수 있어 클래스들을 메모리에서 제거할 수 있다.

Local class table은 자바 프로그램의 메인 스레드(Main thread)가 생성될 때 만들어지며 이후 읽어 들이는 클래스들은 local class table에 추가한다. 프로그램은 자신의 local class table과 global class table을 통해서 필요로 하는 클래스를 찾는다. 그리고 프로그램의 추가적인 스레드(Thread)들은 메인 스레드와 같은 local class table을 공유한다. 그림 6은 3개의 스레드를 가지고 프로그램이 실행되는 경우이다.

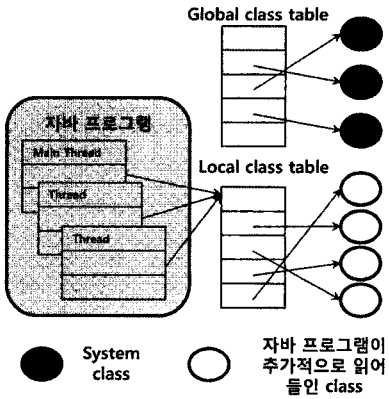

그림 6 Two-level Class Table

4.2 쓰레기 수집기를 이용한 class unloading

쓰레기 수집기는 local class table들을 이용하여 사용하지 않는 클래스들을 쉽게 찾아 수거할 수 있다. 쓰레기 수집기는 살아있는 스레드의 local class table을 방문하여 클래스가 수거되지 않도록 표시하여 종료된 프로그램에서 사용하던 클래스들만 수거되도록 한다. 참고로 global class table에 있는 system class들은 사용중으로 표시하여 자바 가상 머신이 수행 중에는 메모리에 남아 있게 된다.

표준 자바 가상 머신 명세에서는 클래스 로더가 사용되지 않을 때 해당 로더가 읽어 들인 클래스들을 메모리에서 제거할 수 있다고 명세하고 있다. 앞에서 소개한 방법도 프로그램 마다 독립된 클래스 로더를 할당하였다고 볼 수 있다. 추가적으로 표준 자바 가상 머신에서 class unloading을 수행할 때는 static field 등을 고려해야 한다[12]. 하지만 내장형 자바 가상 머신에서는 이러한 문제가 발생하지 않기에 여기서 언급하지 않기로 한다.

5. 구 현

제안한 기법들을 2장에 설명한 자바 시스템에 적용하였다. 쓰레기 수집기는 mark-and-sweep기법[1]과 Table-based compaction 기법으로 구현되어 있다. 개선된 압축 기법이 한번에 압축할 수 있는 compactable area의 개수 M은 실험을 통해 10으로 정하였다. 추가적으로 메모리 관리 기법이 수정되었기에 메모리 할당 정책과 쓰레기 수집기가 호출되는 조건도 수정하여 메모리 단편화가 적게 발생하도록 하였다.

6. 실험 및 평가

6.1 실험 환경

제안된 기법이 구현된 자바 시스템은 실제 동작하는 CDMA 핸드폰 단말기에 적용되었다. 이 단말기는 ARM7 기반의 CPU를 사용하고 있으며 칼라 LCD를 채택하고 있다. 자바 가상 머신을 위해서는 400,000 bytes의 메모리가 할당되었다.

자바 가상 머신과 자바 프로그램의 동작은 단말기에서 확인하였으며 메모리 사용량은 실행 중 메모리 사용량 변화를 측정하기 위해서 에뮬레이터를 이용하여 측정하였다. 실험에서는 배포되는 KVM에 포함되어 있는 6가지 자바 프로그램을 사용하였으며, 게임, 앨범, 주식 거래 등 다양한 종류로 구성되어 있다.

6.2 압축 성능 평가

외부 단편화 문제를 얼마나 극복하였는지를 확인하기 위해서 표 1에서는 6가지의 프로그램을 하나씩 실행 시키는 동안, 압축을 수행했을 때와 수행하지 않았을 때의 가장 큰 빈 영역의 크기와 빈 영역이 몇 개로 나뉘어 존재하는가를 측정하였다.

표 1 압축 기법 성능 평가

	최대 빈 영역의 크기 (bytes)		빈 영역의 개수 (개)	
	압축 안 한 경우	압축 한 경우	압축 안 한 경우	압축 한 있는 경우
MIDPman	25072	254600	557	2
ManyBalls	107276	321016	521	3
WormGame	107276	314568	544	4
PhotoAlbum	79080	297464	537	2
Ticket	107168	272720	510	2
Stock	106624	290964	549	3
평균	88749.3	291888.6	536.3	2.7

압축을 수행하였을 때 빈 영역의 개수가 현저하게 줄어든 것을 알 수 있으며 가장 큰 빈 영역의 크기도 압도적으로 크다는 것을 알 수 있다. 압축을 하지 않았을 때 가장 큰 빈 영역의 크기가 MIDPman을 제외하고는 대부분 100kbytes 부근에서 나왔으며 이는 해당 프로그램이 많은 메모리를 동시에 사용하지 않기 때문이다. 메모리가 400kbytes보다 적었다면 최대 빈 영역의 크기가 이들 프로그램의 경우에도 MIDPman처럼 압축을 수행하였을 때와 비교해서 매우 적게 나올 것으로 예상된다. 압축 기법이 외부 단편화 문제를 상당부분 극복하여 압축을 수행하지 못했을 때와 비교해서 충분히 큰 빈 영역을 얻을 수 있었음을 알 수 있었다.

6.3 Class unloading 성능 평가

아래의 표 2와 그림 7에서 각 프로그램을 실행하고 종료한 후, class unloading을 적용하기 전과 후의 메모리 사용량을 측정 비교하였다.

표 2 Class Unloading 효과

	메모리 사용량 (bytes)		(A)-(B)
	Class Unloading		
	미사용 (A)	사용 (B)	
MIDPman	86720	67012	19708
ManyBalls	76728	65008	11720
WormGame	86176	68316	17860
PhotoAlbum	82336	68524	13812
Ticket	110604	82680	27924
Stock	101424	75736	25688
평균	90664.6	71212.6	19452

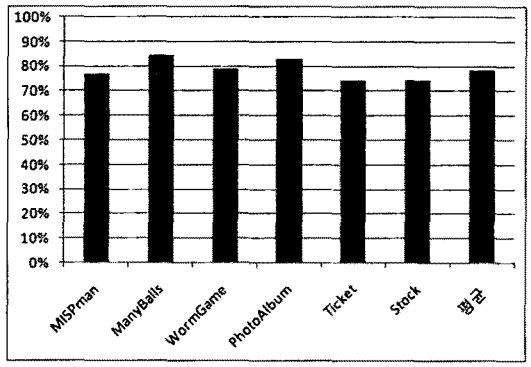

그림 7 Class unloading을 적용한 후의 메모리 사용량

평균적으로 프로그램 종료 후 약 20kbytes 또는 20% 정도의 메모리 사용량이 감소되었다. 프로그램 마다 종료 후 사용 중인 메모리의 사용량이 다른 이유는 클래스 외에도 string table과 같은 기타 자료구조가 프로그램이 종료된 후에도 메모리에 계속해서 남아 있기 때문이다. 하지만 위 실험 결과에서 볼 수 있듯이 class unloading만으로도 메모리 사용량 감소를 얻을 수 있음을 알 수 있다.

다음의 표 3은 6가지 프로그램을 순서대로 모두 한번씩 실행하여 종료하는 실험을 통하여 메모리 사용량을 측정한 결과이다. (A)는 6개의 프로그램들이 모두 종료된 후의 메모리 사용량이며 (B)는 이 상태에서 class unloading을 수행한 후의 메모리 사용량이다. 표를 보면 약 100kbytes의 메모리를 덜 사용함을 알 수 있으며 이는 400kytes의 메모리를 가지는 내장형 시스템에서 매우 큰 메모리 절약을 class unloading만으로도 얻을 수 있다는 것을 뜻한다. 또한 이는 표 2의 결과에서 테스트한 6개에 대해서 프로그램 당 약 20kbytes의 메모리 사용을 절약할 수 있다는 것이 올바른 결과였다는 것을

표 3 6개 프로그램을 모두 수행한 경우

	w/o unloading (A)	unloading (B)	(A)/(B)
6개 프로그램	201,612 bytes	108,212 bytes	0.537

재확인 시켜주고 있다.

마지막으로 6개의 자바 프로그램을 동시에 수행하는 실험을 수행한 결과 압축 기법과 class unloading을 동시에 사용하는 경우에는 동시에 6개의 프로그램을 모두 동시에 수행할 수 있었다. 반면 압축 기법을 사용하지 않는 경우에는 외부 단편화 문제 때문에 큰 메모리 객체를 할당 받지 못하여 6개 모두 동시에 수행하지 못함도 확인하여 외부 단편화 문제를 극복하는 것이 프로그램들을 동시에 수행하는데 필수적인 요소임을 확인하였다. Class unloading 기법의 경우는 프로그램을 동시에 수행할 때에는 효과가 없었으며, 이는 프로그램이 종료된 경우에만 unloading이 수행되므로 당연한 결과이다.

7. 결론 및 향후 계획

본 논문에서는 메모리가 적은 내장형 자바 시스템에서 효과적인 메모리 관리 기법들을 소개하였다. 개선된 압축 기법을 제안하여 외부 단편화 문제를 상당부분 극복할 수 있었으며 분리된 class table을 통하여 쓰레기 수집기가 쉽게 class unloading을 수행하여 메모리를 절약 할 수 있었다. 내장형 자바 시스템에서는 여러 제약 조건 때문에 복잡한 메모리 관리 기법을 도입하지 않고 있지만 간단하면서도 효과가 큰 기법들을 도입하면 그 성능이 많이 향상할 수 있음을 알 수 있었다. 그리고 다음과 같은 방법으로 성능을 더욱 향상시킬 수 있다.

첫째, 본 논문에서는 프로그램이 종료되면 사용되던 클래스들을 메모리에서 제거하는데 동일 프로그램이 실행과 종료를 반복하게 되면 매번 클래스들을 메모리에 읽어 들여야 하기 때문에 수행 속도가 늦어질 수 있다. 선택적으로 class unloading을 적용하면 이와 같은 문제도 해결할 수 있을 것이다. 또한, 현재는 메모리에 남아있는 다른 자료구조들도 class unloading과 비슷한 방법으로 메모리에서 제거할 수 있을 것이다.

둘째, 본 논문에서는 쓰레기 수집기의 기능적인 면을 중점적으로 다루었는데 속도도 중요한 문제가 된다. 대표적인 예로 현재는 전통적인 mark-and-sweep 기법을 사용하고 있는데 개선된 mark-and-sweep 기법[13]을 적용하면 수행 속도를 향상시킬 수 있을 것이다.

마지막으로 메모리가 적은 내장형 시스템을 위하여 제안된 기존 쓰레기 수집 기법들에 본 논문에서 제안한 기법을 적용하면 더욱 큰 효과를 기대할 수 있다. 우선 본 논문과 같은 압축 기반의 기법에 메모리 객체를 압축(compression)하여 Heap 메모리 사용량을 줄이는 기법[14]이 제안되었다. 또한 압축 기법의 문제로 지적되던 긴 정지시간(long pause time) 문제를 세대별(generational) 쓰레기 수집 기법을 이용하여 극복하여 소프트 리얼타임(soft real-time) 제약 조건을 만족시키는 기법[15]이 제안되었다. 하지만 위 연구들은 모든 메모리 영역이 움직일 수 있는 환경에 대하여 연구되었기에 본 논문에서 제안한 압축 기법을 활용하면 메모리가 적은 환경에서 더욱 범용적인 쓰레기 수집기를 구현 할 수 있을 것이라 생각된다.

참 고 문 헌

[1] Jones, R. and Lins, R., Garbage Collection: Algorithms for Automatic Dynamic Memory Management, Wiley, 1996.

[2] Czajkowski, G. and Daynes, L., "Multitasking without compromise: a virtual machine evolution," In Proceedings of the 16th conference on Object oriented programming, systems, languages, and applications, pp. 125-138, 2001.

[3] Connected, Limited Device Configuration Specification, version 1.0, Java Community Process, Sun Microsystems, Inc. http://jcp.org/jsr/detail/30.jsp

[4] KVM Porting Guide, Sun Microsystems, Inc., available as part of the CLDC download package at http://www.sun.com/software/communitysource/j2me/cldc/downlaod.htm

[5] Haddon, B.K. and Waite, W.M., "A compaction procedure for variable length storage elements," the Computer Journal, Vol.10, pp. 162-165, 1967.

[6] Cheney, C.J., "A non-recursive list compacting algorithm," Communications of the ACM, Vol.13, No.11, pp. 677-678, 1970.

[7] Fitch, J.P. and Norman, A.C., "A note on compacting garbage collection," the Computer Journal, Vol.21, No.1, pp. 31-34, 1978.

[8] Lockwood Morris, F., "A time- and space-efficient garbage compaction algorithm," Communications of the ACM, Vol.21, No.8, pp. 662-665, 1978.

[9] Jonkers, H.B.M., "A fast garbage compaction algorithm," Inf. Process. Lett., Vol.9, No.1, pp. 26-30, 1979.

[10] Cohen, J. and Nicolau, A., "Comparison of compacting algorithms for garbage collection," ACM Transactions on Programming Languages and Systems, Vol.5, No.4, pp. 532-553, 1983.

[11] Lindholm, T. and Yellin, F., The Java Virtual Machine Specification, 2nd Ed., Addison Wesley, 1999.

[12] McDowell, C.E. and Baldwin, E.A., "Unloading Java classes that contain static fields," Technical report, University of Calif. Santa Cruz, UCSC-CRL-97-18, 1997.

[13] Chung, Y. C., Moon, S.-M., Ebcioglu K. and Sahlin, D., "Reducing Sweep Time for a Nearly Empty Heap," In Proceedings of the 27th symposium on Principles of programming languages, pp. 378-389, 2000.

[14] Chen, G., Kandemir, M., Vijaykrishnan, N., Irwin, J., Mathiske, B., Wolczko, M., "Heap Compression for Memory-Constrained Java Environments," In Proceedings of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications, pp.282-301, 2003.
[15] Sachindran, N., Moss, J.E.B. and Berger, E.D., "MC2: High-Performance Garbage Collection for Memory-Constrained Environments," In Proceedings of the 19th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications, pp. 81-98, 2004.

최 형 규

2000년 서울대학교 전기공학부 학사. 2002년 서울대학교 전기컴퓨터공학부 석사 2004년 8월~2005년 8월 University of Illinois at Urbana-Champaign 대학원 교환학생. 2002년~현재 서울대학교 전기컴퓨터공학부 박사과정. 주관심분야는 내장형 가상 머신, 컴파일러 최적화, 자바 최적화, 메모리 관리

문 수 묵

1987년 2월 서울대학교 컴퓨터공학과 학사. 1990년 8월 University of Maryland, Computer Science 석사. 1993년 8월 University of Maryland, Computer Science 박사. 1993년 7월~1994년 8월 Hewlett-Packard Co., 소프트웨어엔지니어. 1994년 9월~현재 서울대학교 전기컴퓨터공학부 부교수. 1997년 7월~8월 IBM T. J. Watson 연구소 방문교수 2002년 1월~2003년 2월 Sun Microsystems 방문교수. 주관심분야는 내장형 가상 머신, 컴파일러 최적화, 자바 최적화, 명령어 수준의 병렬처리