메모리가 제한적인 자바가상기계에서의 지역 재사용

(Reusing Local Regions in Memory-limited Java Virtual Machines)

김 태 인 † 김 성 건 † 한 환 수 ††
(Taein Kim) (Seonggun Kim) (Hwansoo Han)

요 약 많은 연구들을 통해 수행 속력, 효율성, 용이성, 안전성을 위하여 메모리 관리 기법들을 개선시켰다. 그러한 방법들 중에서 지역별 메모리 관리 기법은 각각의 객체 할당 위치에서 따라 정해진 지역에 할당 시키고 그 지역이 제거된다면 그 곳에 할당된 모든 객체의 메모리를 반환하는 방법이다. 본 논문에서는 메모리 제약적인 환경에서 힙 메모리 사용량을 줄이기 위해 로컬 지역을 재사용하는 방법을 제시한다. 기본 아이디어는 현재 함수가 수행될 동안 사용하지 않는 상위 로컬 지역을 재사용하는 것이다. 이러한 방법을 사용함으로써 메모리 제약적인 환경에서 메모리의 한계를 극복할 수 있을 것이다.

키워드 : 자바 가상 기계, 지역별 메모리 관리, 지역 재사용

Abstract Various researches had been devoted in purpose of improving memory management in terms of performance, efficiency, ease of use, and safety. One of these approaches is a region-based memory management. Each allocation site selects a specific region, after that allocated objects are placed in this region. Memory is reclaimed by destroying the region, freeing all the objects allocated therein. In this paper, we propose reusing of local regions to reduce heap memory usage in memory-limited environments. The basic idea of this proposal is reusing of upper local regions where objects that are allocated to these regions are not accessed until the current method is finished. We believe our method of reusing local regions is able to overcome memory constraints in memory-limited environments.

Key words : Java Virtual Machine, Region-based Memory Management, Reusing Regions

1. 서 론

임베디드 소프트웨어의 제작과 사용에 있어서 겪게 되는 어려움 중 하나가 바로 메모리 부족 문제이다. 대부분의 임베디드 시스템은 가상 메모리를 지원하지 않고 물리 메모리의 크기도 작은 편이다. 가상 메모리를 지원하지 않는 시스템에서는 가용 메모리가 부족한 경우 실행 중이던 프로그램이 강제로 종료되는 등의 치명적 오류가 발생할 수 있다. 이러한 임베디드 시스템의 메모리 부족 문제를 해결하기 위하여 본 논문에서는 동적 데이타의 지역 할당(region allocation) 및 재사용 기법을 제안하고자 한다.

지금까지 많은 연구자들이 성능, 효율성, 안정성의 관점에서 메모리 관리 기법을 개선시켜 왔다. 그 중 스택 할당 기법은 객체가 생성된 함수에서만 사용될 경우 힙이 아닌 스택에 할당하여 함수가 종료될 때 객체의 메모리를 한꺼번에 같이 반환하도록 한다. 이러한 기법은 평균적으로 보다 빠른 속도로 메모리를 반환하고 공간 재사용 효율을 높여 바람직한 캐쉬 행동 패턴을 야기시킨다. 또한 자바 실행 환경의 성능에 중요한 요소 중에 하나인 가비지 콜렉션의 부담을 줄여준다. 관련 연구에 의하면, 프로그램에 따라 보통 20%에서 최대 70%의 메모리가 자신이 할당된 함수의 수명을 벗어나지 않는 것으로 알려져 있다[1,2]. 그러나 스택 할당 기법은 크기가 큰 배열이나 루프에서 반복적으로 생성되는 객체를 대상으로는 적용하기 힘들다. 이러한 점을 극복하기 위해

·본 연구는 한국과학재단 특정기초연구(R01-2006-000-11196-0)지원으로 수행되었음
† 비 회 원 : 한국과학기술원 전산학과
taein.kim@arcs.kaist.ac.kr
seonggun.kim@arcs.kaist.ac.kr
†† 종신회원 : 한국과학기술원 전산학과
hshan@arcs.kaist.ac.kr
논문접수 : 2007년 2월 6일
심사완료 : 2007년 4월 12일

서 스택이 아니라 지정된 지역을 두어 객체를 할당하는 방법이 지역별 메모리 관리 기법이다[3].

기존의 지역별 메모리 관리 기법들은 메모리 관리를 단순화하여 성능의 향상을 이끌어내는 것을 목적으로 한다. 그러나 이 논문에서 제안하는 지역 재사용 기법은 자원의 제약이 있는 기기에서 성능은 다소 떨어지더라도 메모리를 효율적으로 사용해서 메모리의 부족으로 인한 문제들을 극복하는데 초점을 두고 있다. 데이타 사용 패턴의 가장 큰 특징은 국소성(locality)이므로, 오래 전에 생성된 데이타 중 일부는 현재 위치에서 사용하지 않을 가능성이 높다. 만약 메모리가 부족한 상황이 발생했을 때 앞으로 일정 시간 동안 사용되지 않음이 보장되는 지역이 있다면 해당 지역의 데이타를 2차 저장장치로 이동시킨 후 그 지역을 재사용하여 메모리를 추가로 확보할 수 있게 된다. 이 과정에서 비록 데이타를 저장하기 위한 시간 손실이 발생하지만 그 손실을 최소화하고 공간의 재사용성을 높이는데 목적을 둔다.

이 논문은 다음과 같이 구성되어 있다. 먼저 2장에서 관련 연구를 간략히 살펴본 후 3장에서 본 논문에서 제안하고 있는 자원제약적인 기기들을 위한 지역별 메모리 관리 기법의 전체적인 구조를 소개한다. 그 후 기본적인 힙의 구조 및 객체 할당 방법, 그리고 함수 시작과 종료 시점에서 수행하는 지역 관리 방법에 대해 설명한다. 다음으로 아래방향 레퍼런스를 제거하기 위한 바이트 코드 검사 및 객체 이동방법을 나타낸다. 그리고 4장에서는 지역별 재사용을 위한 조건과 수행 시기 및 수행 방법에 대해서 설명한다. 또한 5장에서 실험 방법과 결과를 살펴보고 마지막으로 6장에서 결론을 도출한다.

2. 관련 연구

Tofte는 정적 프로그램 분석을 통해서 지역별 메모리 관리를 위한 객체의 지역을 추론하는 방법을 제시하였고[4] 이를 바탕으로 ML 프로그램에서 객체를 할당하는 모든 장소마다 분석을 통해 해당되는 지역 정보를 가질 수 있도록 변환시켜 주었다[5]. 이러한 ML Kit 지역 시스템은 지역의 할당과 해제 시점을 자동으로 안전하게 제시해 준다.

C@와 rlang은 언어 수준의 지원을 통해 프로그래머가 직접 지역별 메모리 관리를 적용할 수 있도록 했다[6][7]. C@은 C언어에 지역별 관리를 위한 라이브러리를 추가한 언어이고 rlang은 C@의 확장판이다. 위 방법은 오브젝트의 지역을 추론하기 위한 분석을 하지 않고 각각의 지역마다 레퍼런스 하는 횟수를 헤아려서 지역을 반환할 때 안전성을 보장한다.

일반적인 스택 할당 기법은 정적 분석(escape analysis)을 통해 스택에 할당시킬 수 있는 객체를 찾는다. 그러나 정적 분석 기법은 보통 많은 시간이 소요되고 동적 클래스 로딩이나 실행 시간 컴파일(JIT 컴파일)과 같은 동적인 상황에 대처하기는 힘들다. 그래서 Qian은 객체를 함수 밖에서 사용하는지를 실행 시간에 동적으로 검사하고 적용시키는 방법을 제안하였다[8]. 이 방법은 어떤 함수에 할당된 지역 내에 있는 객체가 함수 범위 밖에서 사용되었을 경우 그 지역을 글로벌 영역화하여 함수가 종료될 경우 지역의 메모리를 해제하지 않는다. 그리고 한 번 함수의 범위에서 벗어난 객체가 생성된 지점에서 다시 객체를 생성할 경우, 다시 함수를 벗어날 가능성이 많다고 판단하여 바로 글로벌 영역에 할당한다. Corry는 이를 좀 더 세분화하여 함수 단위로 지역을 설정하지 않고 루프 단위로 지역을 설정한다[9]. 그리고 객체가 지역의 범위를 벗어나 사용될 경우 지역 전체를 글로벌화 하는 것이 아니라 해당 객체만을 글로벌 지역으로 이동시킨다.

지금까지의 연구들은 메모리의 지역별 관리를 안전하고 효율적으로 하기 위한 방법을 제시하였다. 이 논문에서는 동적인 지역별 관리 기법을 확장하여 지역의 재사용을 통해서 메모리를 절약할 수 있는 방법을 제시하였다. Corry의 방식에서는 아래방향 레퍼런스가 있을 경우 글로벌 지역으로 객체를 이동시키지만 본 논문에서는 그 객체를 레퍼런스하는 로컬 지역으로 객체를 이동시킨다. 그러면 로컬 지역의 비중이 커지게 되고 로컬 지역의 메모리를 재사용할 가능성이 높아진다.

3. 지역별 메모리 관리

지역별 메모리 관리 시스템은 기본적으로 함수마다 각각의 객체 할당 지역을 가진다. 그리하여 함수에서 생성되는 객체들을 해당 함수에 지정된 지역에 할당한다. 그리고 함수가 종료되면 그 지역에 할당된 객체들의 메모리를 바로 반환하여 메모리의 활용성을 높인다. 이렇게 하기 위해서는 반환할 지역에 할당된 객체들이 프로그램에서 더 이상 사용되지 않아야 한다는 조건이 필요하다. 다시 말하면, 정적 변수나 다른 지역에 존재하는 객체들로부터 그것들에게 레퍼런스를 통한 접근을 할 수 없어야 한다. 이런 성질을 만족시키기 위해서는 프로그램 실행 중에 객체를 대입하는 명령어에 대해서 위 조건을 만족하는지 검사를 하고 조정을 해야 한다.

위와 같은 기본 동작에서 가비지 콜렉션 이후 사용 가능한 메모리가 제한 수치보다 작을 경우 지역 재사용 기법이 적용된다. 어떤 함수가 수행되면서 필요한 메모리를 새로 할당하는 것이 아니라 그 함수가 종료될 때까지 사용하지 않는 상위 로컬 지역의 메모리를 재사용한다. 이때 재사용하기 위해서 해당되는 지역의 데이타를 디스크와 같은 다른 저장매체에 기록한다. 그리고 재

사용이 끝나면 다시 원래의 데이타로 복구시킨다.

3.1 힙 구조

그림 1에서 전체적인 힙 구조를 살펴 볼 수 있다. 각각의 함수 마다 하나의 로컬 지역을 가질 수 있고 각 지역은 힙 메모리의 기본단위인 블록들의 리스트로 구성된다. 지역은 크게 글로벌 지역과 로컬 지역으로 나누어 진다. 글로벌 지역은 전역 변수와 같이 여러 함수에서 사용할 가능성이 존재하는 객체들을 할당하는 지역이고 로컬 지역은 그 지역에 대응되는 함수에서만 사용될 객체들을 할당하는 지역이다.

각 지역에 대한 정보는 지역 스택에 저장된다(그림 1 중앙). 지역 스택에 저장되는 정보는 지역의 고유 번호(RegionID)와 블록 포인터 등이다.

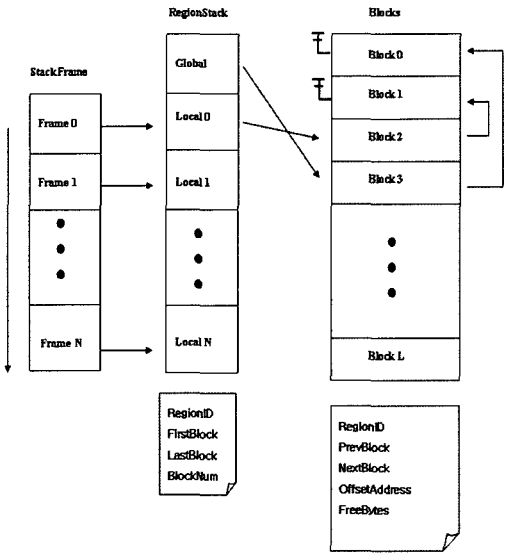

그림 1 힙 구조

로컬 지역에 객체를 할당할 경우 블록의 시작주소부터 차례로 객체 할당에 필요한 크기만큼 확보한다. 하지만 글로벌 지역은 각 블록을 특정 크기의 셀로 나누어 관리하다가 각 객체를 적당한 크기의 셀 하나에 할당한다. 이를 위해 글로벌 지역은 각 크기 별로 사용 가능한 셀의 리스트(FreeList)를 관리하고 있다. 이처럼 글로벌 지역의 메모리를 정해진 크기로 관리하는 이유는 가비지 콜렉션을 통해 사용하지 않는 객체의 메모리를 확보했을 경우 재배치(compaction) 없이 그 공간을 재활용하기 위해서이다. 반면 로컬 지역에 할당된 객체들은 함수가 종료되면 한꺼번에 메모리를 반환하기 때문에 재배치와 관련된 문제가 없다.

함수에서 객체를 글로벌 지역에 할당할 경우 그것의 크기에 해당하는 하나의 셀 메모리를 확보한다. 만약 해당 크기의 빈 셀 메모리가 존재하지 않는다면 글로벌 지역에서 새로운 블록을 하나 할당한다. 그리고 그 블록들을 필요한 셀의 크기로 일정하게 나누어서 사용한다. 일반적으로 글로벌 지역에는 정적 변수들이 할당되고 함수에서 생성된 객체들은 그에 해당하는 로컬 지역에 할당된다.

함수에서 객체를 로컬 지역에 할당할 경우 먼저 그 지역에 해당하는 첫 번째 블록을 찾아 공간을 확보한다. 만약 첫 번째 블록에 충분한 공간이 존재하지 않을 경우 링크를 따라가면서 다른 블록들에 객체를 할당한다. 마지막 블록에도 공간이 충분하지 않을 경우 새로운 블록을 맨 처음에 추가하고 그 곳에 객체 할당을 위한 메모리를 확보하면서 헤더 정보를 생성한다. 새로운 블록을 처음으로 넣는 이유는 다음 차례에 객체를 할당할 경우 바로 공간을 확보할 확률을 높일 수 있기 때문이다. 이러한 과정을 통해서 함수들이 생성하는 객체가 필요로 하는 메모리를 확보할 수 있다.

3.2 지역 생성과 해제

기본적으로 함수가 시작하면 새로운 지역을 생성해서 스택에 쌓고 함수가 종료하면 가장 최근에 쌓인 지역을 제거 하며, 이때 지역을 구성하고 있는 여러 개의 블록들의 메모리도 같이 반환한다.

일반적으로 함수가 시작하면 새로운 스택 프레임이 스택에 쌓이게 되고 새롭게 생성된 지역도 지역을 관리하는 스택에 쌓이게 된다. 로컬 지역을 생성할 때는 고유의 지역 번호만 부여하고 블록을 할당하지는 않는다. 그 이유는 대부분의 함수에서는 객체를 할당하지 않고 단지 여러 가지 값들을 다루기만 하기 때문이다. 그래서 미리 블록을 할당할 필요가 없이 수행 중에 객체를 할당할 공간이 필요하면 그 때 블록을 할당하여 메모리를 확보한다. 이와 같이 지역을 관리하는 일반적인 함수와는 달리 정적 필드를 초기화 하는 <clinit>과 인스턴스 필드를 초기화 하는 <init>함수는 다른 방법으로 지역을 생성한다.

<clinit> 함수는 새로운 지역을 생성하지 않고 글로벌 지역을 사용한다. 이 함수에서 생성한 객체들은 항상 정적 필드를 초기화 하기 위해 사용되기 때문에 글로벌 지역에서부터 생성한 객체의 지역으로, 즉 아래방향의 레퍼런스를 가진다. 그 결과 객체를 글로벌 지역에 할당하지 않는다면 항상 글로벌 지역으로 이동시켜야 되는 불필요한 일들을 하게 된다.

<init> 함수도 새로운 지역을 생성하지 않고 부모 함수의 지역을 사용한다. 이 함수에서 생성한 객체들은 항상 인스턴스 필드를 초기화하기 위해 사용되기 때문에 부모 함수 지역에서부터 생성한 객체의 지역으로 레퍼런스를 가진다. 그 결과 객체를 항상 부모 함수의 지역

으로 이동시켜야 하는 불필요한 일들을 하게 된다.

일반적으로 함수가 종료되면 맨 아래의 스택 프레임과 지역이 제거된다. 이와 함께 지역을 구성하고 있는 블록들의 메모리도 같이 반환한다. 그런데 메모리를 안전하게 반환하기 위해서는 그 지역에 할당된 객체들을 더 이상 사용하지 않아야 하는데 이러한 성질을 만족시키기 위해서 3.3절과 같이 아래 방향의 레퍼런스를 제거해 준다. 다시 말하면, 다른 지역에서부터 반환할 블록에 할당된 객체를 레퍼런스 하지 않아서 함수 종료 후에도 더 이상 사용하지 않기 때문에 안전하게 메모리를 반환 할 수 있게 된다. 안전한 지역 해제를 위해서는 항상 아래의 규칙을 만족시켜야 한다.

- 규칙: 힙에서는 아래방향 레퍼런스가 존재하지 않는다[10].

그림 2의 왼쪽은 아래방향 레퍼런스의 예를 보여주고 있다. 상위 지역에 할당된 객체 X가 그 보다 최근에 할당된 아래쪽 지역의 객체 A를 가리키고 있는 경우 아래방향 레퍼런스가 존재하는 것이다. 이 경우 A가 속한 지역이 해제되면 나중에 X를 통해 A를 접근하려고 할 때 문제가 발생한다. 만약 이러한 아래방향 레퍼런스가 존재하지 않으면 그림 2의 오른쪽에서 나타나듯이 함수 종료 시 상위의 지역에서 가장 아래에 있는 지역에 할당된 객체를 레퍼런스 하지 않기 때문에 함수의 종료와 함께 블록의 메모리를 바로 반환해도 문제가 생기지 않는다[9]. <clinit>와 <init>함수는 새로운 지역을 생성하지 않고 기존에 존재하는 지역을 사용하므로 메모리를 반환할 필요가 없다.

한편, 자바 언어에서 고려해야 할 두 가지 특성이 있다. 처리되지 않는 예외(throw exception)와 finalize

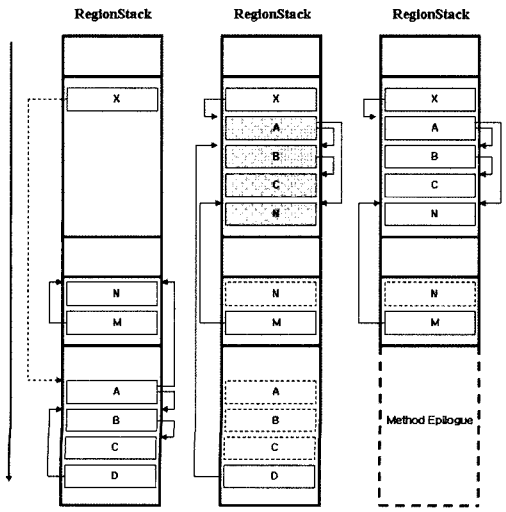

그림 2 아래방향 레퍼런스 제거 및 로컬 지역 반환

함수이다. 실행 도중 처리되지 않는 예외가 발생하였을 경우 함수의 정상적인 종료 과정을 거치지 않고 상위 함수에게 제어권이 넘어가게 된다. 이러한 경우 정상적으로 종료되지 않은 함수의 로컬 지역을 해제해야 한다. 또한 finalize()를 가지고 있는 객체의 할당이 해제될 때에는 그 전에 finalize()를 수행해야 한다. 이 논문에서는 지역 번호를 통해 이러한 객체와 finalize()를 관리한다[8]. 그리하여 함수가 종료될 경우 그 지역의 블록들의 메모리를 반환하기 전에 각 객체의 finalize()를 수행한다.

3.3 객체 이동

로컬 지역을 안전하게 해제하기 위해 3.2절에서 제시한 규칙을 만족시키기 위해서는 실행 도중 아래방향 레퍼런스가 생겼을 경우 이를 제거해야 한다. 다음은 그 과정을 나타내고 있다.

1. 하위 로컬 지역에 할당된 객체와 그것을 레퍼런스 하는 상위 로컬 지역에 할당된 객체를 확인한다.
2. 하위에 할당된 객체와 이를 통해 접근할 수 있는 객체들을 레퍼런스 하는 객체의 상위 로컬 지역으로 이동시킨다. 이때 이동하는 객체들은 그 로컬 지역보다 하위의 로컬 지역에 존재해야 한다.
3. 이동한 객체들의 핸들에서 객체의 주소를 새로운 주소로 갱신한다.

위 과정을 예를 통해 살펴보자. 그림 2의 왼쪽 스택에서 X가 A를 가리키는 경우 규칙을 위배한다. 먼저 X가 할당된 지역과 A가 할당된 지역을 확인한다. 그 후 A를 통해서 접근할 수 있는 B, C, N을 X의 지역으로 이동시킨다. 이때 B, C, N의 지역이 X의 지역보다 하위의 지역임을 확인해야 한다. 만약 하위의 지역에 존재하지 않는다면 규칙을 위배하지 않기 때문에 그 객체는 이동시키지 않는다. 마지막으로 이동을 한 B, C, N 객체를 가리키고 있는 객체 핸들의 주소를 새로운 주소로 갱신한다.

3.4 아래방향 레퍼런스 검사

3.2절의 규칙을 위배하는 경우를 발견하기 위해서 바이트 코드 중에서 객체의 레퍼런스를 저장해 주는 명령어들인 aastore, putfield, putstatic, areturn을 검사해야 한다. 자바에서 객체의 레퍼런스를 저장해 주는 명령어는 위의 4가지가 유일하고 이러한 명령어를 통해서만 규칙을 위배할 수 있기 때문이다[11]. 바이트 코드 중에서 putfield와 aastore는 각각 객체의 필드와 객체 배열에 객체 레퍼런스를 저장하는 명령어이다. 이 두 명령어들을 수행하기 전에 규칙을 위배하는지 검사해야 하고 그 과정은 아래와 같다.

- A.a = B (putfield)
- A[0] = B (aastore)

① B가 null일 경우 통과
② A와 B가 같은 블록에 있을 경우 통과
③ A와 B가 같은 지역에 있을 경우 통과
④ A가 B보다 하위의 지역에 있을 경우 통과
⑤ A가 B보다 상위의 지역에 있을 경우 위배

검사하는 과정에서 오른쪽 객체가 null일 경우, 왼쪽 객체와 오른쪽 객체가 같은 지역일 경우 그리고 위 방향의 레퍼런스가 생성되는 경우는 무사히 통과 시킨다. 왼쪽 객체가 오른쪽 객체보다 상위의 지역일 경우 규칙을 위배하기 때문에 3.3절에서 설명한 과정을 통해서 객체를 이동시켜 규칙을 지키게 된다. 그리고 객체가 속한 지역은 주소를 통해 그 블록의 헤더에 접근해서 알 수 있으며 지역 번호는 로컬 지역이 생성되는 순서에 따라 항상 증가하기 때문에 지역 번호의 크기 비교를 통해 지역간의 상하를 구분할 수 있다.

바이트 코드 중에서 putstatic은 정적 변수에 객체 레퍼런스를 저장하는 명령어이다. 그래서 글로벌 영역에 있는 변수와 객체가 로컬 지역에 할당되어 있는 객체를 레퍼런스 하기 때문에 규칙을 위배할 수 있다. 그래서 명령어를 수행하기 전에 규칙 위배 여부를 검사하는 과정은 아래와 같다.

• ClassA.static_field = obj
① obj가 null일 경우 통과
② obj가 글로벌 지역에 있을 경우 통과
③ obj가 로컬 지역에 있을 경우 위배

검사하는 과정에서 오른쪽 객체가 null일 경우와 글로벌 지역에 할당되어 있을 경우는 무사히 통과 시킨다. 오른쪽 객체가 로컬 지역에 할당되어 있을 경우 항상 규칙을 위배하기 때문에 3.3절에서와 같은 과정을 통해서 객체를 이동시켜 규칙을 지키게 된다.

바이트 코드 중에서 areturn은 객체의 레퍼런스를 부모 함수에 전달하는 명령어이다. 그래서 항상 객체를 부모 함수의 지역으로 이동시켜야 한다. 또 다른 방법으로 부모 함수의 지역과 현재 함수의 지역을 합치면 된다. 전자는 메모리를 아낄 수 있으나 속력 면에서 뒤쳐지고 후자는 그 반대이다. 이 논문에서는 전자의 방법을 사용한다.

4. 지역 재사용

일반적인 호출 스택에서와 같이, 지역별 메모리 관리에서도 먼저 호출된 상위 함수들의 로컬 지역은 현재 수행되고 있는 함수가 실행을 마치고 반환되기 전에는 대부분 사용되지 않는다. 따라서 메모리가 부족한 경우 상위 함수의 로컬 지역을 재사용할 수 있다. 그러나 어떤 상위 로컬 지역이 현재 실행되고 있는 함수의 this 객체나 파라미터 객체들을 통해 접근할 수 있는 경우 그 상위 로컬 지역은 재사용이 불가능하다. 이때 정적 변수들을 포함하지 않는 이유는 3.2절의 규칙을 통해 아래방향 레퍼런스를 제거하는 과정에서 정적 변수들을 통해 접근할 수 있는 객체들은 모두 글로벌 지역으로 이동되기 때문이다. 결론적으로 현재 함수가 수행될 동안 사용할 가능성이 존재하지 않는 상위의 로컬 지역을 재사용 할 수 있다. 즉, 아래의 성질을 만족한다.

• 함수의 this 객체와 파라미터 객체들을 통해서 접근하지 못하는 상위의 로컬 지역은 재사용 할 수 있다.

위 성질은 그 함수의 자식 함수들에게도 마찬가지로 성립한다. 자식 함수들이 상위 로컬 지역의 객체에 접근하기 위해서는 반드시 부모 함수를 통해야 하기 때문이다. 즉, 부모 함수에서 접근 할 수 없는 상위의 로컬 지역은 자식함수들도 접근할 수 없다.

재사용 가능한 로컬 지역을 찾는 과정을 다음 예를 통해 나타내었다. 그림 3과 그림 4는 수행 코드와 그에 따른 힙의 메모리 상태 변화를 표현한다. 코드는 함수 M1이 M2를 호출하고 다시 M2가 M3을 호출하는 상황이다. (A)에서 M1 함수가 시작하면서 로컬 지역을 만들고 생성된 객체 a1, b1을 힙에 할당한다. 그 후 (B)에서 M2 함수가 시작하면서 (A)에서의 과정과 같은 일들이 일어난다. 다음으로 (C)에서는 M3 함수에 필요한 메모리를 힙에 새로 할당하지 않고 M3 함수가 수행되는 동안 사용하지 않는 M1 함수의 지역 메모리를 재사용한다. 그 동안 M1함수의 지역 메모리에 기록된 데이타는 디스크에 기록이 되어 나중에 복구할 때 사용한다. 이와 같이 지역 메모리를 재사용함으로써 메모리 최대 사용량을 줄일 수 있다.

(D)에서는 M3 함수가 종료함으로써 사용한 메모리를 반환하고 기존 상태인 M1 함수의 데이타를 복구한다. (E), (F)에서 M2 함수, M1 함수가 차례로 종료되어 각각의 지역 메모리를 반환한다.

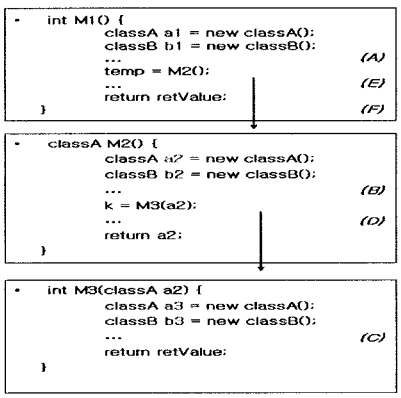

그림 3 예제 코드

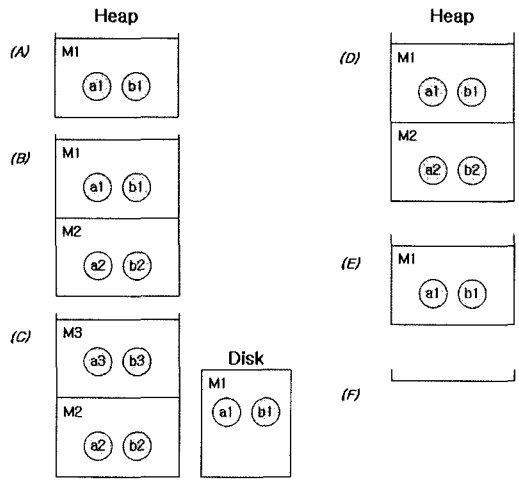

그림 4 힙의 메모리 상태 변화

4.1 지역 재사용 정책

일반적인 자바 가상 기계(Java Virtual Machine, JVM)에서는 힙 메모리가 부족할 때 가비지 콜렉션을 통해서 더 이상 사용하지 않는 객체의 메모리를 반환하여 필요한 공간을 확보하게 된다. 하지만 가비지 콜렉션이 필요한 메모리를 충분히 확보하지 못한다면 프로그램은 메모리 부족으로 수행을 중지한다. 이런 상황에서 일정 시간 동안 사용되지 않는 지역을 재사용한다면 필요한 메모리를 확보할 수 있다.

하지만 지역 재사용을 항상 적용시키기에는 어려운 점이 있다. 재사용을 위해서는 지역에 저장된 데이타를 디스크와 같은 다른 저장매체에 기록을 해서 사용 후 그 원본 데이타를 복구해야 한다. 그런데 디스크에 데이타를 기록하거나 읽기 위해서는 많은 시간적 손실이 발생하기 때문이다. 그래서 지역 재사용 기법은 사용 가능한 메모리가 부족하다고 판단을 할 때 적용해야 한다.

가비지 콜렉션 이후 확보한 메모리가 제한 수치(10%) 이하일 경우 앞으로 메모리 부족이 발생할 것이라 예상하여 현재 함수의 객체 할당부터 지역 재사용 기법을 적용한다. 제한 수치가 커질수록 공간을 재사용할 가능성이 높아지지만 그만큼 시간적 손실이 많이 발생하게 된다. 제한 수치가 작아지면 이와는 반대의 결과가 나온다. 지역 재사용의 적용을 시작한 함수가 종료되었을 경우 디스크에 기록된 데이타를 원래의 지역 블록으로 복구한다. 그 후 메모리가 제한 수치보다 충분하면 지역 재사용 기법을 해제하고 다시 조건을 만족할 때까지 기다린다. 자세한 과정은 4.2절에 나타나 있다.

4.2 지역 재사용 메커니즘

지역을 재사용하기 위한 과정은 아래와 같다.

1. 시작 함수의 this 객체와 파라미터 객체를 통해서 접근 할 수 없는 지역을 찾는다.
2. 메모리를 재사용하기 위해 그 지역에 저장된 데이타를 디스크에 기록한다.
3. 재사용 가능한 지역을 블록 단위로 필요한 함수에 할당한다.
4. 시작 함수가 종료될 때까지 자식 함수들은 위 지역들을 재사용 할 수 있다.
5. 재사용 가능한 지역들을 다 사용했을 경우 새로운 블록을 할당한다.
6. 처음 시작 함수가 종료되면 재사용한 지역들을 복구시키기 위해 블록들의 링크를 처음 상태로 갱신한다.
7. 디스크에 저장된 데이타를 복구한다.
8. 메모리의 용량이 제한 수치보다 충분하면 재사용 동작을 중지하고 아니면 다시 1의 과정으로 돌아간다.

위의 과정에서 나타나듯이 먼저 재사용 가능한 지역들을 찾는다. 그 후 대상 지역들을 재사용하게 되는데 처음 시작하는 함수 내에서 호출한 자식 함수들도 그 대상 지역들을 재사용 할 수 있다. 부모 함수에서 접근할 수 없는 지역들은 자식함수들도 접근할 수 없기 때문이다. 그리고 재사용 가능한 지역들을 다 사용했다면 어쩔 수 없이 새로운 메모리를 할당 받아서 사용해야 한다.

그림 5의 예를 살펴보자. (A)에서 지역 d는 객체를 할당하기 위해 새로운 블록이 필요하다. 하지만 메모리의 여유분이 부족해서 가비지 콜렉션을 호출하게 된다. 그 후 새로 확보한 공간이 4.1절에서의 제한 수치만큼 충분하지 못하다면 지역 재사용 기법을 적용한다. 그림 5에 나타난 수평선은 재사용 기법을 적용하는 경계선이 되고 지역 d를 가진 함수가 시작 함수가 된다. 먼저 지역 d의 함수에서 this 객체와 파라미터 객체들을 통해 접근 가능한 지역을 찾는다. 지역 b는 접근이 가능하고 지역 a, c는 접근이 불가능하다고 가정하면, 결국 지역 a, c의 블록들을 재사용 할 수 있다.

(B)에서 지역 d는 지역 a의 블록을 재사용 하고 지역 e는 지역 c의 블록들을 재사용 한다. 이때 재사용 하기 전에 블록들의 데이타를 디스크에 기록하게 된다. 그 후 함수 e가 종료되면서 사용하던 블록들을 반환하면 지역 c의 블록들을 여유분으로 놓아 둔다. (C)에서 다시 함수 f가 호출되면 재사용 가능한 지역의 블록들을 디스크에 기록하고 사용하는 것이 아니라 우선 여유분으로 있던 지역 c의 블록들을 재사용한다. 다음으로 함수 f가 종료되면 마찬가지로 사용하던 블록들을 반환한다. (D)에서 함수 d가 종료될 경우 시작 함수이기 때문에 지금까지 디스크에 데이타를 기록했던 모든 지역의 블록들을 복

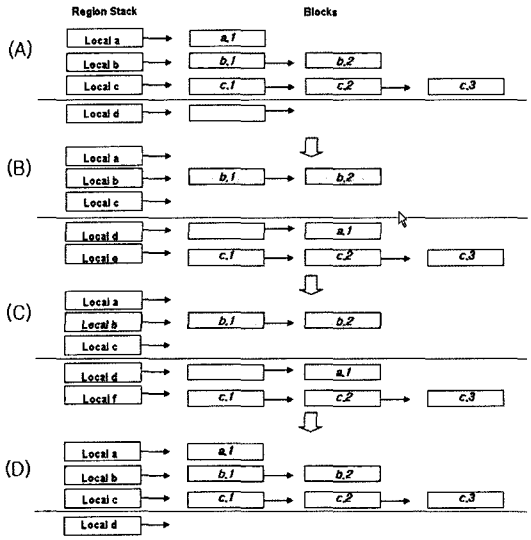

그림 5 지역 재사용 과정

구한다. 그리고 변환된 블록들의 링크를 원래대로 조정한다.

4.3 가비지 콜렉션

지역별 메모리 관리를 적용할 경우 가비지 콜렉션은 글로벌 지역에 할당된 객체들만을 대상으로 하고 로컬 지역에 있는 객체들은 작업 대상에서 제외한다. 로컬 지역에 있는 메모리는 함수가 종료되면 바로 반환이 되기 때문이다.

일반적인 마크-스윕방식의 가비지 콜렉션은 루트 집합을 찾고 그 것을 통해 접근할 수 있는 객체들을 사용하는 것들로 판단하여 표시를 해둔다. 표시되지 않은 나머지 객체들은 더 이상 사용하지 않는 것으로 판단해 메모리를 반환하여 재활용 할 수 있게 된다. 그런데 지역을 재사용 한다면 객체 중 일부가 디스크에 이동됨으로써 객체간의 레퍼런스 관계를 파악하기 힘들어 가비지 콜렉션이 제대로 작동하지 못한다.

이를 해결하기 위해 각 로컬 지역 내의 객체가 글로벌 지역의 객체를 가리키고 있을 경우 해당 글로벌 지역 객체의 레퍼런스를 기록하고 있어야 한다. 이 작업은 3.4절에 소개된 아래방향 레퍼런스 검사 과정 중에 이루어진다. 만약 지역 재사용을 위해 데이타가 디스크에 기록되었을 때 가비지 콜렉션이 불리게 되면 그 해당 지역이 가리키고 있던 글로벌 지역 객체들을 루트 집합으로 포함 시킨다. 이렇게 포함된 객체들 중에는 일반적인 가비지 콜렉션 과정에서는 루트 집합에 포함되지 않을 객체들도 다소 있을 수 있으나 앞으로 사용될 수 있는 객체들의 메모리를 반환하지 않는 것을 간단하게 보장해 줄 수 있다.

5. 실험 결과

5.1 시뮬레이션

본 논문에서 제안한 지역별 메모리 관리기법의 적합성을 검증하기 위해 자바 가상 기계의 객체 할당이나 가비지 콜렉션과 같은 기존의 메모리 관련 동작과 지역별 관리 기법을 수행하는 시뮬레이터를 만들었다. 이것은 벤치마크 프로그램의 메모리 동작을 기록한 파일을 입력으로 받아서 메모리 공간 사용을 시뮬레이션 한다.

벤치마크 프로그램의 메모리 동작에 대한 기록을 얻기 위해 KaffeVM v1.1.7[12]을 이용하였다. 함수의 시작과 종료, 객체의 할당, 객체간의 레퍼런스 대입, 로컬 변수에 객체 레퍼런스 대입 및 오퍼랜드 스택의 변화 등의 사건이 일어날 때 마다 해당 정보를 파일에 기록했다.

시뮬레이터는 메모리 동작을 기록한 파일을 바탕으로 힙, 콜 스택, 오퍼랜드 스택 등의 모든 메모리의 상태를 시뮬레이션 한다. 라이브러리의 네이티브 함수와 같이 자바 가상 기계에서 직접 정보를 알 수 없는 함수가 호출된 경우 파일에 기록되지 않은 메모리 동작이 있을 수 있다. 이 경우에는 시뮬레이터가 직접 각 함수 내에서 일어나는 메모리 관련 작업을 재현하도록 하였다. 예를 들어 `arraycopy()` 함수일 경우, 파라미터로 받는 두 객체의 내용을 복사하는 과정을 추가하였다. 또한 `finalize` 함수를 분석하여 객체가 반환될 때 해야 하는 일들을 추가하였다.

가비지 콜렉션은 4.3절에 설명된 방식의 마크-스윕 알고리즘을 사용했다. 각 로컬 지역이 관리하고 있는 글로벌 지역 객체의 리스트를 루트 집합에 포함하고 로컬 지역은 메모리 반환 대상에서 제외한다. 반면에 글로벌 지역에 있는 마크되지 않는 객체는 메모리 반환 대상이 된다. 객체들 중에서 클래스, 코드, 상수 풀, 메소드 테이블, 정적 변수들은 글로벌 영역에 할당이 되고 항상 루트집합에 포함이 된다.

모든 객체들은 핸들을 통해 관리 된다. 이것은 객체의 주소, 마크 표시, 객체 크기, 지역 번호 정보를 가진다. 객체가 새로운 주소로 이동을 할 때 이 핸들의 주소정보를 갱신한다.

마지막으로 디스크 입출력에 관련 있는 버퍼 캐시를 시뮬레이션 했다. 대개의 운영체제에서는 디스크 입출력을 효과적으로 수행하기 위해 메모리에 버퍼 캐시를 두므로 이를 반영해야 하기 때문이다. 그러나 자원 제약이 있는 환경에서는 버퍼 캐시가 충분하지 않기 때문에 이를 고려해서 버퍼 캐시 크기에 따른 디스크 입출력 시간을 측정했다.

5.2 SpecJVM98

본 실험에서는 SpecJVM98의 각 벤치마크 프로그램

에 대해 지역 재사용 방법을 적용하였다. 지역 재사용 방법을 통해 절약할 수 있는 힙의 크기와 그에 따라 늘어난 실행 시간의 비율을 측정하였다(표 1).

SpecJVM98의 각 벤치마크는 입력 크기를 1, 10, 100으로 구분하여 수행할 수 있다. 본 실험에서는 입력 크기를 1과 10으로 두어 수행했다. 표의 첫 번째 컬럼은 벤치마크의 이름과 사용된 입력의 크기를 나타낸다. 각 컬럼의 RB는 지역별 메모리 관리 방법만을 사용한 경우를 나타내고 RR은 RB에다 지역 재사용 방법을 적용하였을 경우를 나타낸다.

지역 재사용 방법을 사용하면 힙의 크기를 최대 9.3%, 평균적으로 4.2%를 절약할 수 있었다. 즉, 지역별 메모리 관리 방법에 비해 지역 재사용 방법을 적용하면 사용할 수 있는 힙의 크기가 줄어들어도 계속해서 수행할 수 있어 메모리 자원의 한계를 어느 정도 극복할 수 있다. 그러나 실행 시간 증가율을 나타낸 컬럼에서 보듯 지역 재사용 방법을 통해 각 프로그램의 메모리 요구량을 줄일 수 있었으나 프로그램의 실행 시간은 늘어났다.

실행 시간이 늘어난 원인은 크게 4가지로 구분할 수 있다. 1)객체 레퍼런스 대입마다 아래 방향 레퍼런스가 생성되는지 검사, 2)객체를 상위의 로컬 지역으로 이동, 3)재사용을 위한 관리 및 디스크 쓰기와 읽기, 그리고 4)힙의 크기 감소로 인한 가비지 콜렉션의 호출 횟수 증가이다. 지역 재사용 방법을 사용하지 않고 지역별 메모리 관리만 할 경우 실행 시간에 영향을 주는 요소에는 위의 첫 번째와 두 번째 경우만 포함된다.

RB의 경우 평균적으로 0.7% 느려졌는데, 이는 아래 방향 레퍼런스 검사와 객체를 상위의 로컬 지역으로 이동시키는 것은 실행 시간에 큰 영향을 미치지 않는다는 것을 뜻한다. 즉, 지역 재사용 방법으로 인한 실행 시간 증가는 가비지 콜렉션의 발생 횟수와 디스크에 읽고 쓰는 데이타의 양에 따라 크게 좌우된다. 그런데 디스크 관련 오버헤드는 버퍼 캐시의 크기에 따라 많이 달라지기 때문에 이를 고려해야 한다. 자원 제약적인 환경에서는 버퍼 캐시가 충분하지 않기 때문에 본 실험에서는 버퍼 캐시가 256KB일 경우와 0KB일 경우에 대해 실행 시간을 측정했다. 버퍼 캐시가 256KB일 경우 실행시간이 평균적으로 8.87% 증가하였고 0KB일 경우에는 평균적으로 20.37% 증가하였다. 이는 버퍼 캐시를 둘 수 없을 정도로 자원 제약적인 환경에서 평균 20.37%의 시간 오버헤드가 발생하고 사용 가능한 버퍼 캐시가 커질수록 시간 오버헤드가 감소함을 뜻한다. 일반적으로 버퍼 캐시가 없을 경우 모든 디스크 입출력에 대해서 프로세스는 작업이 완료될 때까지 기다려야 한다. 하지만 버퍼 캐시가 존재한다면 디스크에 직접적으로 데이타를 읽거나 기록하지 않고 버퍼를 통해서 작업을 하게 된다. 그렇게 함으로써 프로세스는 디스크에 직접적으로 입출력을 하지 않고 버퍼를 통해서 하기 때문에 시간의 이득을 얻는다. 그런데 jess_10과 jack_1을 제외하면 한번의 지역 재사용 동안 디스크에 쓰고 읽는 메모리 최대 사용량이 버퍼 캐시의 크기인 256KB 보다 작기 때문에 실질적인 디스크 입출력은 일어나지 않는다. 그래서 버퍼 캐시가 0KB일 경우와 256KB일 경우의 차이가 크게 나타난다.

그림 6은 버퍼 캐시의 크기가 0KB일 경우 힙의 크기에 따른 각각의 벤치마크 실행 시간을 추정한 그래프이다. 일반적으로 RB의 경우 힙의 크기가 실행을 위해 필요한 최소 힙의 크기보다 작아진다면 실행 도중에 메모리 부족으로 중지하게 된다. RR은 RB의 최소 힙의 크기보다 더 작은 힙의 크기에서도 벤치마크를 계속 실행할 수 있지만 실행 시간은 증가함을 살펴 볼 수 있다. 대부분의 벤치마크에서 RB의 최소 힙의 크기가 되기 전에는 RB와 RR이 거의 비슷한 움직임을 보인다. 하지만 db_10의 경우는 최소 힙 크기가 되기 전부터 지역 재사용이 일어나 RR의 디스크 관련 오버헤드가 발생하여 차이가 나타난다.

표 1 실험 결과

벤치마크	최소 요구 메모리 (MB)		Ideal w.o. GC (sec)	실행 시간 증가율 (%)			GC (회)		디스크 쓰기량 (MB)
	RB	RR		RB	RR		RB	RR	
					0 (KB)	256 (KB)			
db_1	1.09	1.02(6.6%)	0.61	3.08%	96.44	33.54	1	6	7.25
mpegaudio_1	1.24	1.18(4.5%)	13.26	0.48%	13.73	12.87	2	48	2.48
jess_1	1.76	1.68(4.8%)	3.26	0.77%	17.81	6.50	1	3	8.19
jack_1	2.90	2.79(3.7%)	23.06	0.47%	4.77	1.36	2	2	17.61
db_10	8.14	8.12(0.3%)	15.27	0.37%	10.91	1.53	2	4	30.50
mpegaudio_10	1.30	1.28(1.8%)	134.40	0.02%	0.10	0.04	2	3	0.80
jess_10	2.58	2.34(9.3%)	22.60	0.25%	16.01	14.72	1	46	16.42
jack_10	4.28	4.17(2.5%)	47.43	0.18%	3.20	0.40	1	2	17.92
average		4.2%		0.7%	20.37%	8.87%			

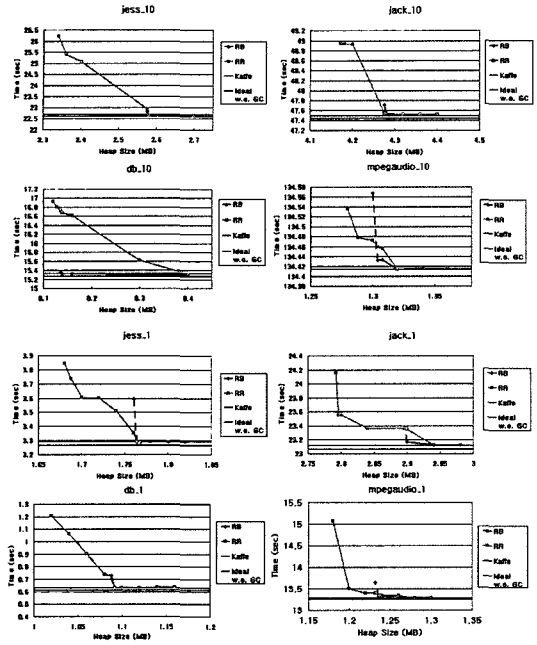

그림 6 힙의 크기에 따른 수행 시간

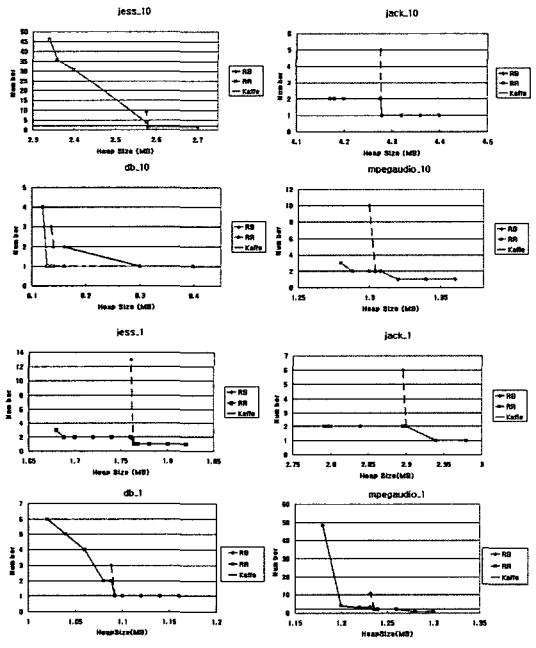

그림 7 힙의 크기에 따른 가비지 콜렉션 호출 횟수

그림 7은 각각의 벤치마크가 힙의 크기에 따른 가비지 콜렉션 호출 횟수를 나타낸 그래프이다. 일반적으로 RB의 경우 힙의 크기가 수행할 수 있는 최소 힙의 크기보다 작아진다면 가비지 콜렉션이 많이 발생하다가

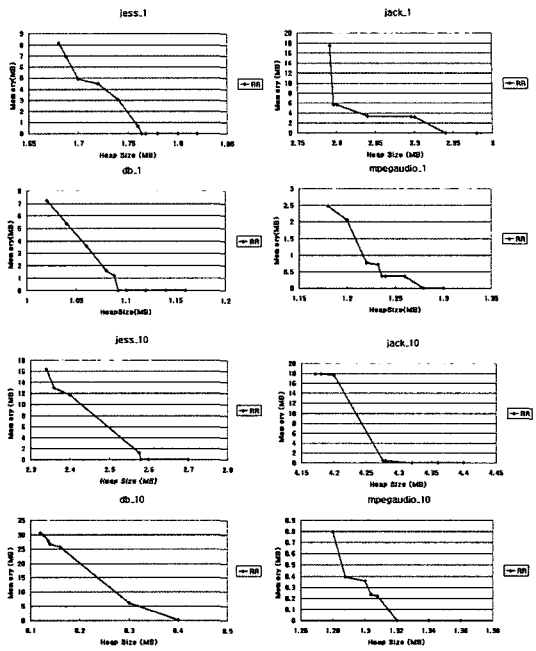

그림 8 힙의 크기에 따른 디스크에 쓴 메모리 양

메모리 부족으로 수행을 중지하게 된다. RR의 경우 RB의 최소 힙 크기보다 더 작을 경우도 계속 수행할 수 있다. 하지만 힙의 크기가 작아진 만큼 가비지 콜렉션의 발생횟수도 증가함을 살펴 볼 수 있다. 일반적으로 RB의 최소 힙의 크기가 되기 전에는 RB와 RR이 비슷한 횟수의 가비지 콜렉션을 호출한다.

mpegaudio_1은 지역 재사용 기법을 적용할 때 필요한 최소 힙 크기에서 갑자기 가비지 콜렉션의 횟수가 늘어난 것에 반하여 jess_10은 힙의 크기가 작아질수록 꾸준히 늘어나는 점을 알 수 있다. 또한 다른 벤치마크들에서는 가비지 콜렉션의 호출이 크게 늘어나지 않고 1 또는 2 정도 증가한다. 이는 대부분의 벤치마크에서 지역 재사용을 통해 메모리를 재사용함으로써 힙의 크기가 작아질 때 증가할 수 있는 가비지 콜렉션 호출을 억제함을 뜻한다.

그림 8은 각각의 벤치마크에서 지역 재사용을 적용한 경우 힙의 크기에 따른 디스크 입출력 양을 나타낸 그래프이다. 일반적으로 RB의 최소 힙의 크기를 기준으로 지역 재사용이 일어나 디스크 입출력이 증가하기 시작하여 힙의 크기가 작아질수록 디스크 입출력이 점점 증가한다. db_10의 경우 RB의 최소 힙의 크기보다 이전에 지역 재사용이 일어나기 시작하는 것을 알 수 있다.

6. 결 론

본 논문에서는 힙 메모리 사용량을 줄이기 위한 지역

재사용 기법을 제안하였다. 제안한 기법의 핵심 아이디어는 특정 함수가 수행될 동안 사용하지 않는 상위의 로컬 지역을 디스크에 옮기고 그 메모리 공간을 재사용하는 것이다. 이를 통해 기존의 방법처럼 필요한 메모리 공간을 새로 할당하는 것이 아니라 재사용을 통해 메모리를 절약 할 수 있다. 하지만 이를 위해서는 추가적인 디스크 입출력이 발생하고 가비지 콜렉션 횟수가 증가한다.

본 논문에서는 시뮬레이터를 통해 SpecJVM98 벤치마크들을 실험하였다. 기존의 지역별 메모리 관리 기법에서 필요한 최소 힙의 크기보다 최대 9.3%, 평균적으로 4.2%정도 줄일 수 있었다. 그에 따른 실행 시간 지연은 버퍼 캐시가 256KB일 때 평균적으로 8.87%가 발생하였고 버퍼 캐시가 없을 경우 평균적으로 20.37%가 발생하였다. 자원 제약적인 환경에서 버퍼 캐시를 256KB정도 사용할 수 있다면 실행 시간 지연이 감소함을 알 수 있고 최대 30% 정도의 지연은 사용자가 프로그램의 예기치 않은 종료 대신 받아들일 수 있는 수준이라 생각된다. 결국, 지역별 메모리 관리 기법에 지역 재사용 기법을 적용하여 프로그램의 메모리 사용량을 줄일 수 있어 자원 제약적인 환경에서 메모리의 한계를 극복할 수 있을 것이다.

참 고 문 헌

[1] J.-D.Choi, M.Gupta, M.J.Serrano, V.C.Sreedhar, S.Midkiff, "Escape Analysis for Java," In Conference on Object-Oriented Programming, System, Languages and Applications (OOPSLA'99), pp. 1-19, 1999.

[2] J.Whaley, M.Rinard, "Compositional Pointer and Escape Analysis for Java Programs," In Conference on Object-Oriented Programming, System, Languages and Applications (OOPSLA'99), pp. 187-206, 1999.

[3] S. Cherem, R. Rugina, "Region Analysis and Transformation for Java Programs," In Proceedings of the 2004 International Symposium on Memory Management (ISMM'04), pp. 131-142, 2004.

[4] M. Tofte, J.P. Talpin, "Region-based Memory Management," Information and Computation, Vol. 132, No. 2, pp. 109-176, 1997.

[5] M. Tofte, L. Birkedal, M. Elsman, N. Hallenberg, "A Retrospective on Region-based Memory Management," Higher-Order and Symbolic Computation, Vol. 17, No. 3, pp. 245-265, 2004.

[6] D. Gay, A. Aiken, "Memory Management with Explicit Regions," In Conference on Programming Language Design and Implementation (PLDI'98), pp. 313-323, 1998.

[7] D. Gay, A. Aiken, "Language Support for Regions," In Conference on Programming Language Design and Implementation (PLDI'01), pp. 70-80, 2001.

[8] F. Qian, L.Hendren, "An Adaptive, Region-based Allocator for Java," In Proceedings of the 2002 International Symposium on Memory Management (ISMM'02), pp. 127-138, 2002.

[9] E. Corry, "Optimistic Stack Allocation For Java-Like Languages," In Proceedings of the 2006 International Symposium on Memory Management (ISMM'06), pp. 162-173, 2006.

[10] H. G. Baker, "CONS should not CONS its arguments, or a lazy alloc is a smart alloc," ACM SIGPLAN Notices, Vol.27, No.3, pp.24-34, 1992.

[11] T.Lindholm, F.Yellin, "The Java Virtual Machine Specification Second Edition," Addison-Wesley, 1999. KaffeVM, http://www.kaffe.org

김 태 인

2005년 한국과학기술원 전산학 학사. 2007년 한국과학기술원 전산학 석사. 2007년~현재 LG 전자 DM 연구소 MAS 그룹 연구원

김 성 건

2003년 한국과학기술원 전기 및 전자공학 학사. 2004년~현재 한국과학기술원 전산학 석박사통합과정

한 환 수

1993년 서울대학교 컴퓨터공학 학사. 1995년 서울대학교 컴퓨터공학 석사. 2001년 University of Maryland, Computer Science PhD. 2001년~2002년 Intel, Architecture Group, Senior Engineer. 2003년~현재 한국과학기술원 전산학과 조교수. 관심분야는 임베디드/모바일 컴퓨팅, 병렬화 컴파일러, 컴퓨터구조