정보학연구 제5권 제2호 2002년 6월(pp.27-34)
The Journal of Information Technology, Vol.5, No.2

CLDC에서 자바 가비지 콜렉션*

권혜은** · 김상훈***

요 약

CLDC를 지원하는 자바가상머신인 KVM의 가비지 콜렉터는 일반적으로 단순한 마크-회수 알고리즘에 기반을 두고 있다. 그러나 이 알고리즘은 단편화 없이 다양한 크기의 객체를 다루기 어렵다는 문제점을 가진다. 본 논문에서는 객체의 크기에 따라 자유 기억 공간의 할당 위치를 결정하는 메모리 할당 방법을 적용한 개선된 마크-회수 가비지 콜렉터를 설계하고 구현하였다. 실험을 통해 단편화 문제가 감소되었고 실행시간의 개선이 있었음을 확인하였다.

Ⅰ. 서론

제한된 자원을 가지는 장비에서의 자바 실행 환경을 정의하는 CLDC(Connected Limited Device Configuration)는 자바가상머신과 핵심 라이브러리로 구성된다.[1] KVM(K Virtual Machine)으로 알려진 이 자바가상머신은 자바가상머신 명세서를 따른다. 자바가상머신에서 자바 클래스 파일을 읽어 바이트 코드를 실행하는데 사용되는 메모리 공간은 자동 메모리 관리 시스템에 의해 관리된다.[2] 가비지 콜렉터로 알려진 자동 메모리 관리 시스템의 기능은 더 이상 사용되지 않는 객체를 발견하고 그 공간을 실행 프로그램에 의해 재사용 하도록 한다. 이로 인해 프로그래머는 프로그램에서 힙 공간에 대한 명백한 할당 및 회수를 담당해야 하는 부담을 가지지 않게 된다.[3]

본 논문에서는 CLDC를 지원하는 자바가상머신을 구성하는 한 요소인 가비지 콜렉터에서 효율적인 메모리 공간 사용에 가장 큰 문제점이 되는 단편화 문제를 최소화하기 위한 할당 방법을 제안한다. 제안된 방법은 객체의 크기에 따라 할당 위치를 결정함으로써 단편화 문제가 개선되어 메모리 낭비가 감소되는 것을 실험을 통해 확인하였다.

본 논문은 2장 관련 연구에서 기존의 가비지 콜렉션 알고리즘 및 자바가상머신에서의 가비지 콜렉션에 관해 살펴본다. 3장은 제안하는 개선된 할당 방법에 관한 설명이고 4장은 기존 알고리즘과의 실험 및 성능평가이다. 5장은 결론 및 향후 연구 과제로 구성되어 있다.

* 본 연구는 한국과학재단의 특정기초연구(과제번호: 1999-1-30300-3)지원에 의한 것임.
** 세명대학교 대학원 전산정보학과
*** 세명대학교 소프트웨어학과 조교수

II. 관련연구

가비지 콜렉터는 응용 프로그램에 의해 더 이상 사용되지 않는 메모리를 회수한다. 이는 프로그램에서 메모리와 관련된 오류의 위험을 상당히 감소시키며[4] 프로그래머가 메모리를 직접 관리해야 하는 부담을 줄여 프로그램의 생산성 향상에 도움을 준다. 이러한 가비지 콜렉터에서 가비지로부터 라이브 객체를 구분하는 방법에는 참조계수(reference counting)와 탐색(tracing) 방법이 있다.[3]

2.1. 참조계수

참조계수 방법에서 객체는 참조 횟수를 기록하기 위한 공간을 가지며 프로그램이 실행되는 동안 참조계수가 변경된다. 객체는 참조계수가 0이 되는 시점에 가비지가 되고 해당 공간은 회수된다. 이는 프로그램이 실행되는 동안 가비지 콜렉션을 위한 연산이 실행되므로 실시간 시스템에 쉽게 적용될 수 있다. 그러나 원형구조를 가진 객체의 회수를 위한 별도의 처리가 필요하며 프로그램의 실행에 비례하는 연산 비용이 소모된다. 또한, 참조계수를 위한 공간낭비의 문제점이 있다.

2.2. 탐색

탐색 방법은 포인터의 탐색을 통해 프로그램에서 사용될 수 있는 객체를 구분 한 후, 이외의 공간을 회수하는 방법이다.

마크-회수(mark-sweep) 기법은 프로그램의 루트 셋으로부터 도달 가능한 모든 객체를 마크하여 라이브객체와 가비지를 구분하고, 마크되지 않은 공간을 회수한다. 이 기법은 단편화 문제없이 다양한 크기의 객체를 다루기 어려운 문제점을 가진다. 콜렉션 비용은 메모리 크기에 비례하며 참조의 지역성 문제도 가진다.

마크-압축(mark-compact) 기법은 도달 가능한 모든 객체를 마크한 후 모든 라이브 객체가 인접하도록 이동시킨다. 이로 인해 마크-회수 기법의 문제점이었던 단편화 문제가 제거되어 다양한 크기를 가진 객체의 할당이 단순해졌다. 그러나 이 알고리즘은 대부분의 객체가 라이브 상태인 경우 마크-회수 기법에 비해 상당히 느린 단점이 있다.

복사(copying) 기법을 사용하는 일반적인 콜렉터는 세미스페이스(semispace) 콜렉터이다. 이는 메모리를 두 개의 인접한 영역으로 분리하고 프로그램이 실행되는 동안 한 영역에 객체를 할당된다. 현재 영역에 할당이 더 이상 불가능한 경우 도달 가능한 모든 객체를 다른 영역으로 복사한다. 객체의 복사로 인해 다양한 크기의 객체를 할당하더라도 단편화 문제가 발생하지 않는다. 수명시간이 긴 객체의 경우 반복적인 복사가 요구되는데 이 문제를 개선한 방법이 세대별 기법이다. 세대별 기법은 메모리를 여러 공간으로 분리하고 객체를 나이에 따라 콜렉션 빈도가 서로 다른 공간에 위치시킨다. 복사기법은 메모리를 분리하여 사용하므로 실제 프로그램에서 필요로 하는 것 보다 많은 공간이 요구되는 단점이 있다.

2.3. JVM의 가비지 콜렉션

SUN사의 자바가상머신인 HotSpot의 가비지

콜렉션은 다양한 방식이 혼합되어 사용된다. 전체 메모리는 세대별 기법을 사용하여 Young영역과 Old영역으로 구분된다. Young영역은 대부분의 객체가 수명이 짧다는 점이 고려되어 하나의 Eden영역과 두 개의 Survivor영역을 이용한 복사 기법으로 관리된다. Young영역에 비해 비교적 큰 Old영역은 단편화 문제가 발생하지 않는 마크-압축 기법을 사용하여 관리되며 선택적으로 Train 알고리즘을 이용하여 점진적으로 가비지를 회수한다. 이러한 다수의 알고리즘은 구현을 위한 오버헤드가 크기 때문에 제한된 환경에 적용되기에는 적합하지 않다.[5]

2.4. KVM의 가비지 콜렉션

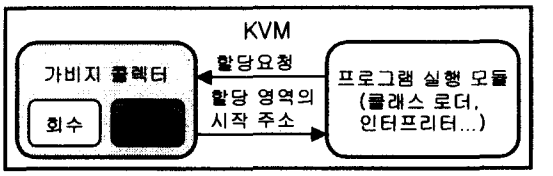
KVM의 가비지 콜렉터에서 사용되는 마크-회수(Mark-Sweep) 알고리즘은 두 단계로 구성된다. 첫 번째 마크 단계는 프로그램의 루트 셋으로부터 도달 가능한 모든 객체를 마크하고, 두 번째 회수 단계는 메모리를 탐색하며 마크 표시가 있다면 표시를 지우고, 표시가 없는 객체가 차지한 공간은 회수한다. 이때 회수된 영역이 인접 하다면 병합된다. 객체는 처음 할당된 이후부터 회수될 때까지 그 위치가 변경되지 않기 때문에 시간이 경과함에 따라 객체들이 분산되어 심각한 메모리 단편화 문제와 참조의 지역성 문제가 발생한다.

III. 단편화 감소를 위한 가비지 콜렉터 개선

프로그램이 시작되고 첫 번째 콜렉션에서 회수되지 않은 대부분의 객체들은 프로그램 종료 시점까지 라이브 상태인 경향이 있다.[5] 그러므로, 본 논문에서는 객체의 위치를 변경하지 않는 마크-회수 방식에서 단편화 문제를 개선하기 위해 객체의 할당 방법을 개선하였다.

3.1. KVM과 가비지 콜렉터

가비지 콜렉터는 콜렉터 자신과 실행 프로그램의 요청에 의해 객체를 할당하는 부분과 더 이상의 할당이 불가능한 경우 루트 셋으로부터 도달 불가능한 공간을 회수하는 부분으로 구성된다. 콜렉션 이후에도 할당 불가능한 경우 실행 프로그램은 종료된다.



[그림 1] KVM과 가비지 콜렉터

[그림 1]은 KVM내의 다른 모듈들과 가비지 콜렉터의 관계를 나타낸다. 바이트 코드를 실행하기 위해 필요한 메모리 공간은 가비지 콜렉터 내의 할당기를 통해 할당된다. 할당을 위한 공간을 찾지 못한 경우 가비지의 회수를 위한 모듈이 호출된다.

3.2. 할당 가능한 영역의 구성

할당기는 이중 링크드 리스트로 구성된 할당 가능 영역과 리스트 양끝의 첫 번째 노드를 지시하는 두 개의 포인터로 구성된다. 할당 영역은 영역 구분 점에 의해 두 개의 영역으로 구분되며 영역 구분 점의 위치는 할당 요청의 변화에 따라 가변적이다. 리스트의 각 노드는 자신의 크기를 나타내는 부분과 이전, 이후 노드를 지시하는 포인터로 구성되어 서로 연결되며 할당 요청에 따라 노드의 정보는 변경된다.

3.3. 할당 요청에 대한 처리

할당기는 프로그램 실행 모듈들로부터 할당 요청을 받아 올바른 요청인지 확인하고 크기에 따른 할당 함수를 호출한다. 할당 함수는 자신의 할당 시작 포인터를 이용하여 리스트의 노드를 탐색한다. 적합한 노드를 발견한 경우 링크를 조절한 후 노드의 위치를 반환하고 그렇지 않는 경우 회수 담당 함수가 호출된 후 다시 할당을 시도한다. 가비지 콜렉션 이후에도 적합한 공간을 발견하지 못하면 OutOfMemoryError가 발생하고 프로그램의 실행은 즉시 중단된다.

3.4. 할당 함수의 내부 구조

[표 1]은 할당기의 알고리즘이다. 개선된 할당기는 고유의 할당 시작 위치를 가지는 mallocFrontAlloc와 mallocBackAlloc으로 구성된다. KNIFE에 지정된 것 보다 작은 요청은 첫 번째 노드에서 마지막 노드 방향으로 할당되는 mallocFrontAlloc에서 처리되고 큰 요청은 마지막 노드에서 첫 번째 노드 방향으로 할당되는 다른 함수에서 처리된다. 함수는 할당 시작 위치를 나타내는 포인터를 이용하여 동일하거나 보다 큰 노드를 탐색하다가 동일한 크기의 노드를 발견한 경우 해당 노드를 할당하고, 큰 노드를 발견한 경우 노드는 분할된다.

[표 1] 할당 알고리즘

```
cell* allocateFreeChunk(long 요청크기) {
  return (요청크기 < KNIFE) ?
  mallocFrontAlloc(요청크기) :
     mallocBackAlloc(요청크기);
}
cell* mallocFrontAlloc(long 요청크기) {
  CHUNK thisChunk = 리스트의 왼쪽 시작 위치;
  while(thisChunk) {
        .....
     if(현재 공간의 크기 > 요청크기) {
       할당위치 = 현재공간시작주소
       현재공간시작주소 += 요청크기
        .....
     } else if(현재 공간의 크기 == 요청크기) {
           현재 위치 할당
        .....
  }
        .....
     thisChunk = thisChunk->next;
  }
 return NULL;
}
cell* mallocBackAlloc(long 요청크기) {
  CHUNK thisChunk =
  리스트의 오른쪽 시작위치;
    while(thisChunk) {
```

```
        .....
            할당위치 = 현재공간의 마지막주소
                        - 요청크기
        .....
    thisChunk = thisChunk->prev;
  }
  return NULL;
}
```

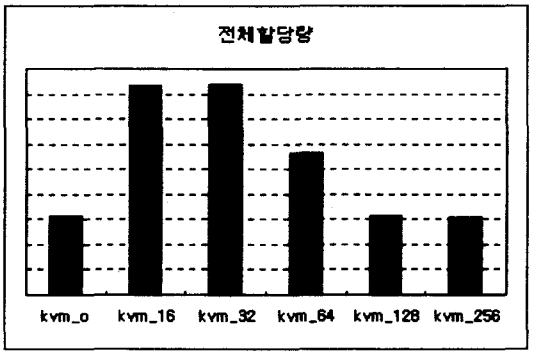
[그림 4]는 각 함수에서 할당 가능한 영역을 분할하여 사용할 필요가 있는 경우에 분할 전과 후의 구성이다. 첫 노드부터 할당되는 malloc FrontAlloc의 경우 노드의 시작위치부터 할당이 시작되어 할당 가능한 영역의 시작주소가 변경된다. mallocBackAlloc의 경우 노드의 마지막 부분부터 할당되므로 할당 가능한 영역의 시작 주소는 할당이 완료되어도 변함없다.

〔그림 2〕 할당 가능한 영역의 분할

IV. 실험 및 성능 평가

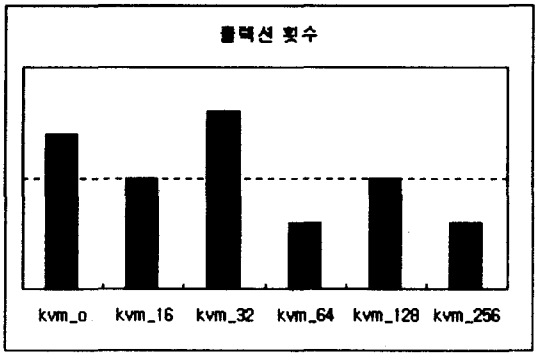
실험환경은 SUN SPARC Ultra-60 하드웨어에 Sun OS 5.7, gcc 와 cc를 이용하였다. 메모리는 150KB로 제한하고 기존 알고리즘(kvm_o)을 이용하는 경우와 16(kvm_16), 32(kvm_32), 64(kvm_64), 128(kvm_128), 256(kvm_256)개의 셀을 기준으로 개선된 할당 알고리즘을 적용한 경우에 대해 실험하였다.

다음은 서로 다른 수명시간을 가지는 최대 512셀, 응용 프로그램에서의 평균 할당 요청은 340셀, 실행에 필요한 모든 할당을 포함한 전체 평균 73셀의 객체가 할당되는 경우에 대한 실험 결과이다.



〔그림 3〕 전체 할당량

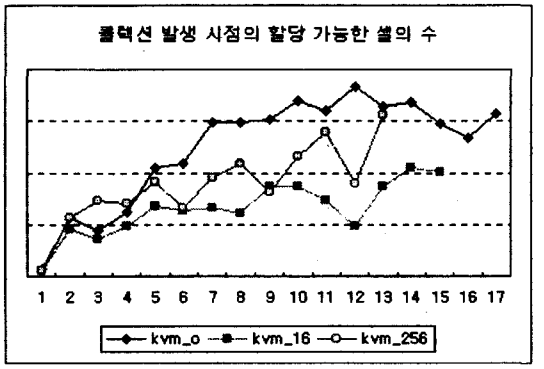
[그림 5]는 메모리 부족 에러가 발생하는 시점까지의 전체 할당량을 나타낸다. 32셀을 기준으로 구분하여 할당했을 경우 기존 알고리즘 보다 10,456셀 더 많이 할당되었다. 최악의 경우는 256셀을 기준으로 구분하여 할당한 경우로 기존 알고리즘보다 140셀 적게 할당되었다.



〔그림 4〕 콜렉션 횟수

[그림 6]은 메모리 부족 에러 발생 시점까지

발생한 전체 콜렉션 횟수를 나타낸다. 기존 알고리즘에 비해 콜렉션 횟수가 가장 작은 경우는 kvm_64와 kvm_256의 경우이다. 두 경우 콜렉션 발생 횟수는 동일하지만, 전체 할당량은 kvm_64의 경우가 kvm_256보다 5,093셀 더 많이 할당되었다.



[그림 5] 콜렉션 발생 시점의 할당 가능한 셀의 수

[그림 7]은 전체 할당량만을 고려했을 때 기존 알고리즘과 기존 알고리즘에 비해 성능이 가장 우수한 kvm_16, 비교적 성능이 떨어지는 kvm_256의 할당기를 이용하여 콜렉션이 발생한 시점에서 남아있는 셀의 수를 나타낸다. 그래프의 끝 지점은 더 이상의 할당이 불가능하여 OutOfMemoryError가 발생되고 프로그램이 종료된 시점이다. 세 경우 모두 동일한 회수 알고리즘이 적용되어 유사한 형태의 그래프를 나타내고 있다. 그러나 기존 알고리즘에 비해 제안된 알고리즘이 콜렉션이 발생하는 시점에서 할당 가능한 셀의 수가 더 적다. 즉, 더 많은 셀이 할당된 후 콜렉션이 발생되었음을 알 수 있다.

V. 결론 및 향후 연구 과제

본 논문에서는 제한된 환경에서 자바 프로그램을 실행하는 KVM의 가비지 콜렉터의 단편화 문제를 최소화하기 위해 객체의 크기에 따라 할당 위치를 결정하는 방법을 제안하고 실험을 통해 성능을 확인하였다. 실험 결과 다양한 요청에 대해 콜렉션 횟수, 전체 할당량, 할당 불가능한 시점의 남은 메모리 양을 측정했을 때 대부분의 경우 기존 알고리즘보다 높은 성능을 나타내었고, 특히 16셀과 32셀 단위로 객체 크기를 구분하여 할당했을 때 성능이 우수하였다.

현재의 KVM의 가비지 콜렉터에서 일반적으로 사용되는 마크-회수 방식은 프로그램의 할당 요청을 더 이상 수용할 수 없는 경우 사용자 프로그램이 중단되고 가비지 콜렉션을 실행한다. 이는 처리에 대한 마감시간이 존재하는 실시간 시스템에서는 적합하지 않은 방식이다. 그러므로 실시간 시스템에서의 가비지 콜렉션을 위한 부분을 향후 연구과제로 하여 보완할 것이다.

참고문헌

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

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

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

[4] Tobias Ritzau, Hard Real-Time Reference Counting without External Fragmentation, ETAPS, 2001

[5] Sun Microsystems, Tuning Garbage Collection with the 1.3.1 Java Virtual Machine

[6] Sun Microsystems, Support Readiness Document Java 2 Platform, Micro Edition: Connected Limited Device Configuration 1.0.2, Mobile Information Devece Profile 1.0.1, and J2ME Wireless Toolkit 1.0.1, 2001

[7] Sun Microsystems, Connected, Limited Device Configuration specification Version 1.0a, 2000

[8] Eric Giguere, Java 2 Micro Edition, WILEY 2000

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

Java Garbage Collection in CLDC

He-Eun, Kwon* · Sang-Hoon, Kim**

Abstract

The KVM garbage collector implemented in CLDC was generally based on the simple mark-sweep algorithm, but it is difficult to handle objects of varying size without fragmentation of the available memory. In this paper, we have designed and implemented a memory allocator based on the mark-sweep algorithm that minimizes the fragmentation by the method that determines the allocation position of free-space list according to object size. The experimental result shows that our algorithm reduce the fragmentation and improve the execution time than the existing algorithm.

* Dept. of computer & information, Semyung Univ.
** Dept. of software, Semyung Univ.