공간 조정을 이용한 복사 메모리 재사용 알고리즘

우 균, 김성청[0], 주성용, 차미양*, 한태숙**
동아대학교 컴퓨터공학과, *동아대학교 교육대학원 정보컴퓨터교육전공, **한국과학기술원 전자전산학과
woogyun@mail.donga.ac.kr, kkk2947@hanmail.net, jheaven1@kornet.net, chmiyang@hotmail.com, han@cs.kaist.ac.kr

Adjusting the Semi-Spaces in a Copying Collection

Gyun Woo, Soung-Choung Kim[0], Sung-Yong Joo, Mi-Yang Cha*, Tae-Sook Han**
Dept. of Computer Engineering, Dong-A University,
*Major in Information Computer Education, Graduate School of Education and training, Dong-A University,
**Division of Computer Science, Dept. of Electrical Engineering and Computer Science, KAIST

요 약

이 논문은 메모리 재사용 알고리즘 중 하나인 복사 알고리즘의 공간 오버헤드를 완화시키기 위한 방법을 기술한다. 복사 알고리즘은 연속적인 메모리 공간을 확보하는 전통적인 메모리 재사용 알고리즘으로, 노드 크기가 일정하지 않은 추상 기계의 구현에 널리 사용되는 알고리즘이다. 그러나, 복사 알고리즘은 전체 힙 공간의 반을 복사 프로세스를 위해서 남겨두어야 하기 때문에 공간 오버헤드는 100%에 달한다. 이 논문에서는 공간 오버헤드를 완화시키는 기본 아이디어를 제시하고, 이 기본 아이디어에서 힙 경계 부분을 처리하기 위한 세 가지 방법을 제안한다.

1. 서 론

메모리 재사용 시스템(garbage collector)[1]은, 메모리 관리를 자동으로 지원하는 현대 프로그래밍 언어에 있어서 핵심적인 부분이다. 전통적인 메모리 재사용 알고리즘으로는 표시-검사(mark-sweep) 알고리즘, 참조수 계산(reference counting) 알고리즘, 복사(copying) 알고리즘이 있는데[2], 이 중 복사 알고리즘은 연속된 메모리 공간을 확보할 수 있다는 면에서, 힙 노드 크기가 일정하지 않은 추상기계[3, 4]에 널리 사용되고 있다.

그러나 복사 알고리즘은 공간 오버헤드가 100%에 달한다는 단점이 있다. 이는 복사 알고리즘이 현재 사용하고 있는 힙 공간과 같은 크기의 힙 공간을 복사할 공간으로 확보하기 때문이다. 그러나 사실 현재 사용 중인 힙 공간과 같은 크기의 복사할 공간을 확보할 필요는 없으며, 현재 사용하고 있는 힙 공간 내의 유효 자료에 해당하는 공간만을 확보하면 된다.

본 논문은 이에 착안하여 복사할 힙 공간의 크기를 100%보다 작은 비율로 확보하는 방법을 제시한다. 최초의 메모리 재사용 호출 시에는 기존 알고리즘과 같은 방식으로 수행하고, 이로부터 유효 그래프 비율을 예측한 후, 이 값에 따라 복사할 힙 공간의 비율을 사용 중인 힙 공간의 100%이하로 줄인다.

본 논문의 구조는 다음과 같다. 먼저 2절에서 관련 연구를 몇 가지 살펴본다. 3절에서는 이 논문에서 제안하는 복사 알고리즘의 기본 아이디어와 힙 경계를 처리하기 위한 세 가지 방법을 제안하고, 이들 경계 처리 방법의 수행속도를 분석한다. 4절에서는 이 논문에서 제안한 복사 알고리즘의 한계에 대하여 간단히 논한다. 마지막으로 5장에서 결론을 맺는다.

2. 관련 연구

복사 알고리즘은 처음 제안[5, 6]된 이후 많은 개선이 이루어져 왔는데, 특히 Baker에 의해 많은 개선 방법이 연구된 바 있다. 1978년 Baker가 제안한 점진적 방법[7]은, 일반적인 순차 컴퓨터(serial computer)에서 메모리 할당과 동시에 메모리 재사용을 수행함으로써, 메모리 재사용을 위해 컴퓨터 수행이 멈추는 현상을 개선하고 있다. 이 방법은 재사용 시스템의 공간 특성도 어느 정도 개선하고 있는데, Baker 자신도 이 알고리즘의 공간 특성 개선은 미미하다고 밝히고 있다.

1991년 Baker는 복사 알고리즘을 변형한 방법으로 트레드밀(Treadmill) 방법[8]을 제안하였다. 이 방법은 모든 사용 가능한 노드를 원형 이중 연결 리스트(circular doubly linked list)로 연결한 후, 직접 노드를 복사하는 대신 링크 연결을 변경함으로써 복사 알고리즘을 수행한다. 이 방법은, 언뜻 보기에는 별도의 복사 공간을 필요로 하지 않기 때문에 공간 특성을 많이 개선한 듯 보이나 실제로는 이중 연결 리스트 유지에 필요한 포인터가 각 노드마다 두 개씩 필요하기 때문에 여전히 공간 부담이 크다.

3. 공간 조정을 이용한 복사 알고리즘

3.1 기본아이디어

공간 조정을 이용한 새로운 복사 알고리즘의 기본 아이디어는, 복사 알고리즘에서 메모리 재사용(garbage collection: 이하 GC) 수행이 완료된 후 유효 자료의 총 크기[1]가 복사를 위해 마련해 두었던 공간보다 일반적으로 작다는 데 있다. 물론 최악의 상황을 가정한다면 현재 사용 중인 힙 공간(FromSpace)과 같은 크기로 복사를 위한 공간(ToSpace)을 확보해야 하겠지만, 일반적으로 이런 최악의 상황은 자주 발생하지 않는다. 만약 이러한 최악의 상황이 발생한다 하더라도, 이는 GC 부담이 큰 경우이므로 메모리 공간을 더 확보하는 것이 바람직할 것으로 생각된다.

복사 알고리즘에 따라 GC가 수행되는 일반적인 상황을 도시하면 그림 1과 같다. 그림 1은 전체 힙의 왼쪽 반공간(semi-space)을 사용하던 중, 힙 공간이 부족하여 오른쪽 반공간으로 복사가 일어나는 상황을 나타낸다. 즉 왼쪽 공간이 FromSpace이고 오른쪽 공간이 ToSpace이다.

그림 1에서 HeapStart와 HeapEnd는 전체 힙의 경계를 나타내는 상수이다. FromSpace의 시작과 끝을 나타내는 FromHeap과 HeapBound, ToSpace의 시작을 나타내는 ToHeap은

1) 이를 해당 프로그램 그래프의 레지던시(residency)라고 한다. 프로그램 수행 시점에서 프로그램의 레지던시란, 해당 시점에서 유효한 그래프(live graph) 크기이다.

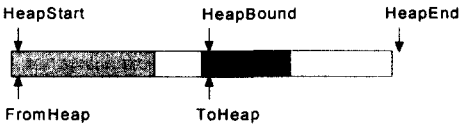

그림 1 복사 알고리즘 수행 상황

매 GC 수행 후에 변경되는 변수이다.

이 논문에서는, 이러한 힙 구조의 양끝을 붙인 힙을 가정한다. 즉 그림 2와 같은 원형 구조의 힙을 가정한 후, FromSpace와 ToSpace의 크기 비율을 자유롭게 조정한다.

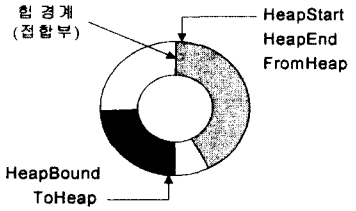

그림 2 가상적인 원형 힙 구조

그림 3은 그림 2와 같은 원형 구조의 공간에서 FromSpace와 ToSpace의 크기 비율을 조정하는 알고리즘을 나타낸다. 그림 3은, C와 유사한 형태의 Johns와 Lins 표기법[1]을 따랐는데, 함수 정의에 사용되는 =과 구분하기 위해 치환 연산자는 기호 :=를 사용하였다.

```
GC(n) =
    일반적인 복사 알고리즘 수행;
    /* liveRatio 계산 */
    liveRatio := (복사된 graph 크기)/(FromSpace 크기);
    liveRatio += Threshold;
    FromSpace, ToSpace 크기 재 조정;
```

그림 3 공간 조정을 이용한 복사 알고리즘

그림 3 알고리즘의 핵심은 일반적인 복사 알고리즘을 수행한 후, 이전의 FromSpace 공간 크기에 대한 FromSpace의 유효 그래프 크기 비율인 liveRatio를 산출하여, FromSpace와 ToSpace의 크기를 조정하는 것이다. 상수 Threshold는 ToSpace 크기에 여유를 두기 위한 상수이다.

3.2 경계처리 알고리즘

그림 3에 따라 GC를 수행하면 특정 반공간이 힙 경계에 걸쳐 놓일 수 있으므로 이 경우를 처리해 주어야 한다. 이 논문에서는 세 가지 경계처리 알고리즘을 제시한다. 경계처리 알고리즘을 살펴보기 전에, 먼저 기존 복사 알고리즘의 메모리 할당 방법을 살펴보면 그림 4와 같다.

```
메모리 할당 부분:
[ChkFree(n); (Alloc(word);)^n]*

ChkFree(n) = if (! Enough(n)) GC(n);
Enough(n) = return (Hp+n < HeapBound);
Alloc(word) = *Hp++ := word;
```

그림 4 기존 복사 알고리즘

그림 4에서 메모리 할당 부분이란, 일반적인 프로그램 수행 시 메모리 할당이 일어나는 부분을 설명한 것이다. 여기서 위첨자는 해당 부분의 반복 횟수를 나타내는데, 즉 그림 4의 메모리 할당 부분은 항상 n 개 워드를 할당할 수 있는 공간이 있는가 검사(ChkFree(n))한 후, 해당 워드를 실제로 할당하여 초기화(Alloc(word))하는 과정의 반복임을 나타낸다.

가장 단순한 경계처리 알고리즘은, 경계 부분을 위한 검사를 매번 수행하는 단순한 방식으로, 그림 5와 같다.

```
Enough(n) =
    if (HeapBound < FromHeap)
        return (Hp+n-HeapSize < HeapBound);
    else
        return (Hp+n < HeapBound);
Alloc(word) =
    *Hp++ := word;
    if (Hp >= HeapEnd) Hp := HeapStart;
```

그림 5 경계처리 알고리즘 1

그림 5에서 메모리 할당 부분과 ChkFree 함수는 그림 4의 기존 복사 알고리즘과 같다. Enough 함수는 먼저 현재 사용중인 FromHeap 공간이 걸쳐져 있는 공간인가 검사한 후, 각 경우에 따라 충분한 공간이 있나 검사한다. Alloc 함수는 매 워드를 초기화 때마다 Hp가 힙 범위를 넘었나 검사하여, Hp가 힙 범위를 넘은 경우 Hp를 HeapStart로 재조정한다.

그림 5의 알고리즘은 워드 할당 때마다 Hp가 힙 경계에 있는가 검사한다는 단점이 있다. 사실 이 작업은 Hp가 힙의 끝 부분에 위치한 경우에만 필요하며, 이에 대한 정보는 CheckFree에서 얻을 수 있다. 이러한 방식의 경계처리 알고리즘 2를 기술하면 그림 6과 같다.

```
메모리 할당 부분:
[if (ChkFree(n) == NORMAL) (Alloc(word);)^n
else (BoundAlloc(word);)^n]*

ChkFree(n) =
    (isEnough, flag) := Enough(n);
    if (isEnough) return flag;
    else    GC(n);
            (isEnough, flag) := Enough(n);
            return flag;
Enough(n) =
    if (HeapBound < FromHeap)
        return (Hp+n-HeapSize < HeapBound,
            (Hp+n >= HeapEnd)? BOUND: NORMAL);
    else return (Hp+n < HeapBound, NORMAL);
BoundAlloc(word) =
    *Hp++ := word;
    if (Hp >= HeapEnd) Hp := HeapStart;
```

그림 6 경계처리 알고리즘 2

그림 6에는 Alloc과 BoundAlloc 두 개의 메모리 할당 함수가 있는데, ChkFree의 결과에 따라 이 중 하나를 선택한다. 이를 위해 Enough 함수도 변경되었는데, Enough 결과의 두 번째 필드가 현재 경계 부분(BOUND)인지 아닌지(NORMAL)를 나타낸다.

그림 6의 알고리즘은 현재 할당하는 위치가 경계인가 아닌가 여부에 따라 다른 방식으로 메모리 할당을 수행하므로 메모리 할당 부담이 적다. 그러나 메모리 할당 코드가 두 배로 늘어난다는 단점이 있다. 그러나 만약 경계 부분에 놓이는, 조각난 공간(조각난 공간 크기 < 노드 크기)을 포기한다면 알고리즘은 더 간단해 질 수 있다. 이를 기술하면 그림 7과 같다.

```
Enough(n) =
    if (HeapBound > FromHeap)
        return (Hp+n < HeapBound);
    else if (Hp+n < HeapEnd)
        return true;
    else if (n < HeapBound-HeapStart)
        Hp := HeapStart;
        return true;
    else
        return false;
```

그림 7 경계처리 알고리즘 3

그림 7에 기술되어 있지 않은 부분(메모리 할당 부분과 ChkFree, Alloc)은 그림 4와 같다. 사실 그림 7의 알고리즘은 Enough 함수만 제외하면 기존 복사 알고리즘과 동일하다. Enough 함수는 현재 할당하고자 하는 메모리 영역이 경계에 있는 경우, Hp를 변경함으로써 기존 Alloc 함수를 그대로 사용할 수 있도록 한다.

3.3 경계처리 알고리즘 비교

이 절에서는, 몇 가지 기본 연산의 수행시간을 가정하여, 이상에서 설명한 세 가지 경계처리 알고리즘의 수행 속도를 예측해 본다. 먼저 몇 가지 비율을 정의하겠는데, ChkFree 호출 시점에 FromSpace가 힙의 경계에 걸쳐져 있게 되는 경우의 비율을 R, ChkFree 시점에 할당할 워드 수 n과 현재 힙 포인터 Hp를 합한 값이 힙의 경계인 HeapEnd를 넘어가는 경우의 비율을 B라고 하자. 그리고 각 산술, 비교, 저장 연산의 수행 시간을 a, c, m이라고 하자. 이러한 가정 하에 각 알고리즘의 ChkFree와 Alloc 함수의 수행 시간을 계산하면 표 1과 같다.

표 1 알고리즘 수행속도 분석

알고리즘	ChkFree	Alloc
기존알고리즘	a+c	a+m
알고리즘 1	a+c+Ra	a+m+c+Bm
알고리즘 2	a+3c+2m+R(2a+c)	a+m+B(c+m)
알고리즘 3	a+2c+R(a+c)+Bm	a+m

a: 산술 연산 시간(arithmetic operation time)
c: 비교 연산 시간(comparison operation time)
m: 저장 연산 시간(move operation time)
R: FromSpace가 경계에 걸쳐진 비율(reverse configuration ratio)
B: (Hp+n > HeapEnd)인 비율(bound condition ratio)

표 1에 의하면 알고리즘 1~3의 Alloc 속도가 모두 저하되었지만, 사실 알고리즘 2의 Alloc 속도는 기존 알고리즘의 Alloc 속도와 큰 차이가 없다. 비율 R은 FromSpace가 힙 경계에 걸쳐져 있는 경우를 나타내므로 이는 꽤 높은 값일 수 있다. 그러나 비율 B는 그렇게 높은 값이 아니다. 예를 들어, 워드 하나가 2바이트이고 그래프 노드 하나의 평균 워드 수가 8, 전체 힙 공간이 8MB(megabyte)라고 하면, B는 대략 2×10^{-16}에 해당한다. 따라서 알고리즘 2의 Alloc 수행 속도는 기존 알고리즘과 거의 비슷하다고 볼 수 있다.

4. 공간 조정 복사 알고리즘의 한계

이 논문에서 제안한 공간조정 복사 알고리즘은 크게 두 가지 한계가 있다. 하나는 복사를 통한 메모리 재사용 함수 호출 시 더 많은 메모리 부족 오류가 발생할 수 있다는 점이다. 예컨대 어떤 프로그램의 수행될 때, 프로그램 레지던시가 전체 힙 공간의 절반에 가깝고, 이러한 레지던시가 장시간 지속된다면, 이 프로그램에 대하여 본 논문의 알고리즘은, 레지던시에 가까운 상황에서 메모리 재사용 시스템이 호출될 때, 메모리 부족 오류를 발생시킬 수 있다.

본 논문의 알고리즘의 두 번째 한계는, 기존 복사 알고리즘보다 기본 연산의 수행 속도가 늦다는 점이다. 그러나 본 알고리즘을 사용하면 공간 조정을 통해 전체적인 메모리 재사용 함수의 호출 횟수를 줄일 수 있으므로 본 논문에서 제안한 방법이 반드시 느리다고 단정지을 수는 없다.

5. 결 론

이상에서 공간 조정을 이용한 복사 메모리 재사용 알고리즘을 기술하였다. 공간 조정에서 필수적으로 대두되는 경계 조건 처리 문제를 해결하기 위해 세 가지 방법을 제시하였고, 간단한 분석을 통해 각 방법의 수행 속도를 비교하였다. 보다 구체적인 속도 비교를 위해서는 이들 알고리즘을 구현하여 실험해 봐야 하는데, 이는 향후 연구로 남겨둔다.

참고문헌

[1] Jones, R. and Lins, R., *Garbage Collection — Algorithms for Automatic Dynamic Memory Management*, John Wiley & Sons, 1996.

[2] Wilson, P. R., "Uniprocessor Garbage Collection Techniques," *In Proceedings of International Workshop on Memory Management*, pages 1-34, September 1992.

[3] Peyton Jones, S. L., "Implementing lazy functional languages on stock hardware: the Spineless Tagless G-machine," *Journal of Functional Programming*, 2(2): 127-202, 1987.

[4] Gyun, W. and Taisook, H., "Compressing the Graphs in G-machine by Tag-Forwarding," *Journal of KISS (B)*, 26(5):703-713, May 1999.

[5] Fenichel, R. R., and Yochelson, J. C., "A LISP Garbage Collector for Virtual Memory Computer Systems," *Communications of the ACM*, 12(11): 611-612, November 1969.

[6] Cheney, C. J., "A Non-recursive List Compacting Algorithm," *Communications of the ACM*, 13(11):677-678, November 1970.

[7] Baker Jr, H. G., "List Processing in Real Time on a Serial Computer," *Communications of the ACM*, 21(4): 280-294, April 1978.

[8] Baker Jr, H. G., "The Treadmill: RealTime Garbage Collection without Motion Sickness," *In OOPSLA '91 Workshop on Garbage Collection in Object-Oriented Systems*, October 1991. Also appears as *SIGPLAN Notices*, 27(3):66-70, March 1992.