논•문

공간 조정을 이용한 복사 메모리 재사용 알고리즘의 성능 평가

(Evaluating the Performance of Adaptive Copying Collector)

김 성 청, 우 균

동아대학교 컴퓨터공학과

kkk2947@hanmail.net, woogyun@mail.donga.ac.kr

요 약

복사 메모리 재사용 시스템은 그 알고리즘이 간단하고 메모리 압축(compaction)을 지원하는 장점으로 인해 널리 사용되는 메모리 재사용 시스템이지만, 공간 오버헤드가 100%에 달한다는 단점이 있다. 복사 메모리 재사용 시스템의 공간 오버헤드를 100% 이하로 줄이는 방법 중 하나로 공간 조정에 의한 복사 메모리 재사용 시스템이 제안된 바 있다. 이 방법은, 메모리 재사용 시스템이 호출될 때 유효 그래프의 실제 크기가 현재 사용하는 공간보다 작으므로, 힙의 절반을 복사를 위한 공간으로 확보할 필요가 없다는 것에 착안한 방법이다. 이 논문은 공간 조정에 의한 복사 메모리 재사용 시스템의 구현과 실험결과를 기술하고, 세 가지 경계처리 알고리즘의 수행 성능을 비교한다. 실험결과에 따르면, 경계처리 알고리즘 3을 사용하였을 때의 수행 성능이 가장 좋은 것으로 나타났다.

1. 서론

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 확

본 논문은 2001년도 정보통신부 IT관련학과 장비지원사업의 동아대학교 대응자금에 의해 연구되었음

보하기 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.

이러한 특성을 이용하여 복사 알고리즘의 공간 오버헤드를 개선하기 위한 알고리즘으로, 공간 조정을 이용한 복사 메모리 재사용 알고리즘[5,6]이 제시된 바 있다. 그러나, 이 알고리즘 성능은 분석적인 방법을 통해 추정되었을 뿐이며 구체적인 실험을 통한 성능 평가가 이루어진 바 없다. 본 논문에서는 공간 조정을 이용한 복사 메모리 재사용 알고리즘을 구현하고 기존 복사 알고리즘에 대한 이 알고리즘의 성능을 구체적인 실험을 통해 평가해 보고자 한다.

본 논문의 구성은 다음과 같다. 2절에서는 관련연구로 본 논문에서 구현하여 성능을 평가하고자 하는 공간 조정을 이용한 복사 메모리 재사용 알고리즘에 관해 기술한다. 3절에서는 공간 조정을 이용한 복사 메모리 재사용 알고리즘의 구현 및 기존 복사 알고리즘과 개선된 알고리즘의 성능 평가에 대해 기술한다. 그리고 4절에서 결론과 향후 연구 과제에 관하여 기술한다.

2. 관련연구

공간 조정을 이용한 복사 알고리즘의 기본 아이디어는, 기존 복사 알고리즘에서 메모리 재사용 수행이 완료된 후 유효 자료의 총 크기가 복사를 위해 마련해 두었던 공간보다 일반적으로 작다는 데 있다. 따라서, 메모리 재사용 시스템의 호출 시점에서 해당 프로그램의 유효 자료의 총 크기인 레지던시(residency)를 고려하면, 현재 사용공간(FromSpace)에 비해 복사를 위한 공간(ToSpace)의 크기 비율을 작게 조정할 수 있다.

공간 조정을 이용한 복사 알고리즘은 다시 공간 조정 부분과 경계처리 부분으로 나뉜다. 공간 조정 알고리즘은 레시던시를 고려하여 FromSpace와 ToSpace의 크기 비율을 조정하는, 기본 틀이 되는 알고리즘이고, 경계처리 알고리즘은 현재 사용중인 공간이 힙의 경계에 걸쳐져 있을 때 메모리 할당을 처리하는 알고리즘이다.

2.1 공간 조정 알고리즘

공간 조정 알고리즘은 가장 최초에 메모리 재사용 시스템을 호출할 때에는 기존 복사 알고리즘을 수행하고, 이로부터 프로그램의 공간 특성을 예측하여 ToSpace의 비율을 FromSpace의 100%이하로 줄이는 것이다. 공간 조정 알고리즘을 개략적으로 나타내면 그림 1과 같다.

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

그림 1 공간 조정 알고리즘

그림 1 알고리즘은 일반적인 복사 알고리즘을 수행한 후, 이전의 FromSpace 공간 크기(CurSize)에 대한 FromSpace의 유효 그래프 크기 비율인 liveRatio를 산출하여,

FromSpace와 ToSpace의 크기를 조정하는 것이다. 상수 Threshold는 liveRatio의 급격한 변화에 대비하여 ToSpace 크기에 여유를 두기 위한 상수로서, 실제 구현에서는 힙 전체 크기의 10%로 하였다.

2.2 경계처리 알고리즘

공간 조정 알고리즘을 이용하여 From-Space와 ToSpace 크기 비율을 조정하다 보면 FromSpace가 힙 경계에 걸쳐지는 경우가 발생한다. 이 때, 경계 부분에 대한 메모리 할당을 처리해 주어야 하는데, 그림 2는 경계처리가 필요한 상황을 나타내고 있다.

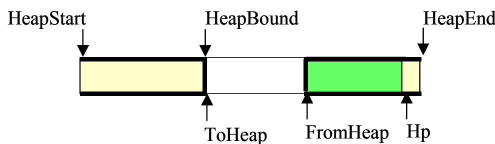

그림 2 경계처리의 필요성

그림 2에서 FromSpace는 굵은 선으로 표시된 부분이다. 이 때, 할당하고자 하는 노드의 크기가 힙의 경계(HeapEnd)와 현재 힙 포인터(Hp) 사이의 크기보다 더 큰 경우 부족한 만큼을 힙의 첫 부분(HeapStart)에 쓸 수 있게 해야 한다. 이를 처리하기 위한 방법이, 경계처리 알고리즘으로, 세 가지 경계처리 알고리즘이 제시된 바 있다.

경계처리를 위한 첫 번째 알고리즘인, 경계처리 알고리즘 1은 경계 부분을 위한 검사를 매번 수행하는 단순한 방식이다. 경계처리 알고리즘 1은 매 워드를 초기화 할 때마다 Hp가 힙 범위를 넘었는지 검사하여, Hp가 힙 범위를 넘은 경우 HeapStart로 재조정한다. 이 알고리즘은 알고리즘을 이해하기는 쉬우나, 매 워드를 초기화 할 때마다 Hp를 검사하므로 그 수행부담이 클 것으로 예상된다.

경계처리 알고리즘 2는 충분한 공간이 있나 검사할 때 Hp의 위치 정보를 얻어, Hp가 힙의 끝 부분에 위치하였는가 여부에 따라 메모리 할당을 다르게 처리한다. 할당하고자 하는 노드가 힙 경계에 걸쳐지게 되면 경계처리 알고리즘 1에서와 같은 메모리 할당 방법을 사용하고, 그렇지 않으면 일반적인 메모리 할당 방법을 사용한다. 경계처리 알고리즘 2는 경계처리 알고리즘 1에 비해 Hp 검사 부담이 줄어든다는 장점이 있지만, 메모리 할당 코드가 두 배로 늘어나는 단점이 있다.

경계처리 알고리즘 3은, 경계 부분에 놓이는 조각난 공간(조각난 공간 크기 < 노드 크기)을 포기하는 알고리즘이다. 경계처리 알고리즘 3은 현재 할당하고자 하는 노드가 힙의 경계에 걸쳐지게 된다면, 노드를 경계에 위치시키지 않고 힙의 처음 부분에 할당하는 알고리즘이다.

3. 성능 평가

3.1 구현 및 실험 환경

알고리즘 구현을 위해 사용한 컴파일러는 ZG-Machine의 실험에 사용된 G-Machine Compiler이다. 이 컴파일러는 Haskell[7] 및 C[8] 언어로 작성되었으며 메모리 재사용 알고리즘은 Cheney의 복사 알고리즘[9]을 사용하고 있다.

이 컴파일러는 초조합자(super combinator)[10]로 주어진 원시프로그램을 G-코드

(G-code: G-machine에서 사용되는 코드)로 번역하고, 이를 다시 M-코드[11] 형태로 변환시킨 후, 이 M-코드로부터 최종 C 코드를 생산한다. 이들 변환 부분은 모두 Haskell로 작성되었다. 생산된 C 코드와 메모리 재사용 시스템을 포함한 실행 환경 코드는 C 컴파일러를 통해 최종 실행파일로 번역된다. 본 논문에서는 최종 실행파일을 수행하여 메모리 재사용 시간을 비교함으로써 새로 제안된 메모리 재사용 시스템의 성능을 평가하였다.

실험을 위해 두 가지의 시스템을 사용하였는데, 사용된 시스템의 상세 내역은 표 1과 같다.

표 1 실험을 위해 사용된 시스템

	CISC 기계	RISC 기계
운영체제	RedHat Linux 6.0	SunOS 5.7
프로그램	Pentium-Ⅲ/667	Sun Sparc
메모리	128MB	512MB

실험을 위해 두 가지 시스템을 사용한 이유는 CISC(Complex Instruction Set Computer)에서의 실험 결과와 RISC(Reduced Instruction Set Computer)에서의 실험 결과를 비교해 보기 위해서이다.

실험을 위해 사용 된 프로그램은 nofib 벤치마크[12]의 imaginary 부분에서 가져온 것인데, exp, nfib, queens, primes, tak 등 모두 5개의 프로그램을 가지고 실험하였다.

3.2 실험 결과

실험 결과는 프로그램의 전체 수행 시간, 메모리 재사용 시스템 수행 시간, 메모리 재사용 시스템의 호출 횟수 등 세 가지 기준으로 측정하였고, 실험 결과로 사용된 값은 G-machine의 기본 메모리를 2MB로 주어, 다섯 개 프로그램을 각각 5회 수행한 후 평균한 값이다. C 코드로부터 최종 실행파일을 만들기 위해 사용한 C 컴파일러는 GNU C 컴파일러(CISC: gcc version 2.96, RISC: gcc version 2.95.2)이고, 최적화 단계 2 옵션(-O2)을 적용하여 실행파일을 생성하였다.

3.2.1 전체 수행 시간 비교

기존 복사 알고리즘과 공간 조정을 이용한 복사 알고리즘에 경계처리 알고리즘 1, 경계처리 알고리즘 2, 경계처리 알고리즘 3을 각각 구현한 후, 프로그램 전체 수행 시간을 측정하여 비교하였다.

경계처리 알고리즘 1을 이용한 경우 전체 수행 시간은 기존 복사 알고리즘에 비해 CISC 기계에서 평균 39%, RISC 기계에서 평균 31% 증가한 것으로 나타났다. 경계처리 알고리즘 1은 매 워드를 초기화 할 때마다 Hp가 경계에 위치하였는가 여부를 검사하는데, 이 검사 부담 때문에 전체 수행 시간이 증가한 것으로 생각된다.

경계처리 알고리즘 2를 채택한 경우 역시 전체 수행 시간이 모든 프로그램에서 증가한 것으로 나타났는데, 경계처리 알고리즘 2의 다섯 개 프로그램의 전체 수행 시간은 기존 복사 알고리즘에 비해 CISC 기계에서 평균 29%, RISC 기계에서 평균 27% 증가하였다. 경계처리 알고리즘 2를 적용한 경우 수행 속도는 경계처리 알고리즘 1을 적용하였을 때보다 기존 알고리즘에 비해 CISC 기계에서는 평균 12%, RISC 기계에서는 평균 3% 정도 빨라진 것으로 나타났다. 그러나 경계처

리 알고리즘 2도 기존 알고리즘에 비해 많이 느린데, 이는 경계처리 알고리즘 2가 Hp의 위치에 따라 메모리 할당을 다르게 하도록 메모리 할당 함수를 선택하는 부분의 수행 부담이 크기 때문이라 생각된다.

경계처리 알고리즘 3을 사용한 경우에는 모든 프로그램의 전체 수행 시간이 감소했다. 이 경우 다섯 개 프로그램의 전체 수행 시간은 기존 복사 알고리즘에 비해 CISC 기계에서는 평균 26%, RISC 기계의 수행 시간은 평균 25% 감소하여, 경계처리 알고리즘 3이 세 개의 경계처리 알고리즘 중 전체 수행 시간이 가장 빠른 것으로 나타났다.

표 2는 경계처리 알고리즘 3을 사용한 경우 전체 수행 시간을 기존 복사 알고리즘을 사용하였을 경우와 비교한 것이다. 전체 수행 속도 증가율을 백분율로 나타내었는데, 증가율은 다음 식에 의해 계산되었다.

$$\text{증가율} = \frac{Tt_3 - Tt_0}{Tt_0}$$

여기서, Tt_3는 경계 처리 알고리즘 3을 이용한 경우 전체 수행 시간이고, Tt_0는 기존 복사 알고리즘을 이용한 경우 전체 수행 시간이다.

3.2.2 메모리 재사용 시스템 수행 시간 비교

기존 복사 알고리즘에 대한 공간 조정을 이용한 복사 알고리즘에 경계처리 알고리즘 1, 경계처리 알고리즘 2, 경계처리 알고리즘 3을 각각 구현한 후, 메모리 재사용 시스템 수행 시간을 측정하여 비교하였다.

경계처리 알고리즘 1을 채택한 경우 메모리 재사용 시스템 수행 시간은 기존 복사 알고리즘을 사용하였을 때에 비해 CISC 기계에서 평균 29%, RISC 기계에서 평균 25% 감소되었다.

경계처리 알고리즘 2를 채택한 경우 메모리 재사용 시스템 수행 시간은 기존 복사 알고리즘을 사용하였을 때에 비해 CISC 기계가 평균 22%, RISC 기계에서 평균 20% 감소되었다.

경계처리 알고리즘 3을 채택한 경우 메모리 재사용 시스템 수행 시간은 기존 복사 알고리즘을 사용하였을 때에 비해 CISC 기계가 평균 32%, RISC 기계에서 평균 28% 감소되어, 세 가지 경계처리 알고리즘 중 경계처리 알고리즘 3이 프로그램의 전체 수행 성능뿐만 아니라 메모리 재사용 시스템의 수행 성능 또한 향상시킴을 알 수 있다.

표 2 기존 복사 알고리즘과 경계처리 알고리즘 3의 전체 수행 시간 비교

(단위: 1/100초)

	CISC 기계			RISC 기계		
	기존	경계처리3	증가율(%)	기존	경계처리3	증가율(%)
exp	1564.6	1366.8	-13	4985.4	4194.0	-16
nfib	1918.2	1427.6	-26	6669.0	5143.4	-23
primes	35.6	27.6	-22	114.4	92.0	-20
queens	2343.2	1549.8	-34	8031.0	5500.6	-32
tak	1884.4	1217.0	-35	6670.4	4523.0	-32
평균	-	-	-26	-	-	-25

표 3은 경계처리 알고리즘 3을 채택한 경우 메모리 재사용 시스템 수행 시간을 기존 복사 알고리즘의 경우와 비교한 것이다. 각 벤치마크 프로그램에 대하여 메모리 재사용 시스템 수행 속도 증가율을 백분율로 나타내었는데, 증가율은 다음 식에 의해 계산되었다.

$$증가율 = \frac{Tg_3 - Tg_0}{Tg_0}$$

여기서, Tg_3는 경계 처리 알고리즘 3을 이용한 경우 메모리 재사용 시스템 수행 시간이고, Tg_0는 기존 복사 알고리즘을 이용한 경우 메모리 재사용 시스템 수행 시간이다.

3.3.3 메모리 재사용 시스템 호출 횟수 비교

기존 복사 알고리즘과 공간 조정을 이용한 복사 알고리즘의 메모리 재사용 시스템 호출 횟수를 비교한 결과가 표 4에 나타나있다. 기존 복사 알고리즘과 공간 조정을 이용한 복사 알고리즘의 메모리 재사용 시스템 호출 횟수 증가율을 백분율로 나타내었다. 증가율은 다음 식에 의해 계산되었다.

$$증가율 = \frac{Ng_{new} - Ng_{old}}{Ng_{old}}$$

여기서, Ng_{new}는 공간 조정을 이용한 복사 알고리즘을 이용한 경우 메모리 재사용 시스템 호출 횟수이고, Ng_{old}는 기존 복사 메모리 알고리즘을 이용한 경우 메모리 재사용 시스템 호출 횟수이다.

표 4 메모리 재사용 시스템 호출 횟수 비교

	기존	공간조정	증가율(%)
exp	530	303	-42.8
nfib	643	358	-44.3
primes	8	5	-37.5
queens	545	313	-42.6
tak	773	430	-44.4

표 4에서 볼 수 있는 바와 같이 메모리 재사용 시스템의 호출 횟수가 가장 많이 감소한 프로그램은 tak로서 44.4% 감소하였으며, 가장 적게 감소한 프로그램은 primes로 37.5% 감소하였다. 다섯 개 프로그램의 메모리 재사용 시스템의 호출 횟수 감소율은 평균 42.3%에 달함을 알 수 있다. 이는 공간 조정을 통해 FromSpace에 대한 ToSpace의 크기 비율을 100%이하로 줄인 결과이다.

표 3 기존 복사 알고리즘과 경계처리 알고리즘 3의 메모리 재사용 수행 시간 비교

(단위: 1/100초)

	CISC 기계			RISC 기계		
	기존	경계처리3	증가율 (%)	기존	경계처리 3	증가율 (%)
exp	235.2	116.6	-50	342.8	243.4	-29
nfib	3.6	2.8	-33	12.8	9.6	-25
primes	2.4	2.0	-17	5.4	3.6	-33
queens	125.8	87.8	-30	323.4	208.9	-36
tak	18.4	12.6	-32	49.6	41.0	-17
평균	-	-	-32	-	-	-28

3.4 실험의 한계

실험에 사용된 각 벤치마크 프로그램의 평균 레지던시를 살펴보면 표 5와 같다. 평균 레지던시는, 메모리 재사용 시스템을 호출할 때마다 레지던시 비율을 계산하여 합한 후, 메모리 재사용 시스템의 호출 횟수로 나눈 것이다.

표 5 각 프로그램의 평균 레지던시

	레시던시(%)
exp	6.61
nfib	0.16
primes	6.17
queens	6.95
tak	0.62

표 5에서 볼 수 있는 바와 같이, 실험한 프로그램의 평균 레지던시가 매우 작다. 본 논문에서 사용한 벤치마크 프로그램은 아주 작은 프로그램이므로, 이러한 경향이 보다 현실적인 대형 프로그램에서도 나타날 것이라고 확신할 수는 없다. 작은 프로그램에 대해서만 실험한 이유는 실험에 사용된 컴파일러가 초조합자를 원시 프로그램으로 받는다는 제한 때문이다. 더 정확한 결과를 얻기 위해서는 본 논문에서 사용한 컴파일러와 같은 실험용 컴파일러가 아닌 실제 컴파일러에 구현하여 다양한 프로그램들에 대한 실험을 수행하여야 한다.

4. 결론 및 향후 연구

본 논문에서는 복사 알고리즘의 공간 오버헤드를 개선하기 위해 공간 조정을 이용한 복사 메모리 재사용 알고리즘의 구현과 실험에 대하여 기술하였다. 구체적인 실험을 통하여 공간 조정 메모리 재사용 시스템의 세 가지 경계처리 방법의 속도를 비교하였고 공간 조정을 이용한 복사 메모리 재사용 알고리즘의 성능을 평가하였다.

실험 결과에 따르면, 공간 조정 복사 알고리즘의 세 가지 경계처리 방법 중 세 번째 방법인 경계처리 알고리즘 3을 적용한 경우 프로그램의 전체 수행 속도와 메모리 재사용 시스템의 수행 속도가 가장 빠른 것으로 나타났다. 기존 복사 알고리즘의 공간 오버헤드를 줄인 결과로, 메모리 재사용 시스템의 호출 횟수를 평균 44.4% 줄일 수 있었고, 이에 따른 메모리 재사용 시스템의 수행 시간도 평균 30.0% 줄일 수 있었다. 그 결과 프로그램 전체 수행 시간에 있어서도 수행 시간이 평균 25.5% 감소하였다.

본 연구에서는 실험용 컴파일러를 사용하였기 때문에 보다 현실적인, 다양한 벤치마크 프로그램에 대하여 실험을 수행할 수 없었다. 좀 더 정확한 결과를 얻기 위해 실험용 컴파일러가 아닌 실제 컴파일러에 구현하여 다양한 예제를 통해서 실험하여 성능을 평가해 보아야 하는데, 이는 향후 연구 과제로 남겨둔다.

참고문헌

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

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

1992.

[3] S. L. Peyton Jones, “Implementing lazy functional languages on stock hardware: the Spineless Tagless G-machine,” *Journal of Functional Programming* 2(2), pp127-202, 1987.

[4] G. Woo and T. Han, “Compressing the Graphs in G-machine by Tag-Forwarding,” *Journal of KISS (B)*, 26(5):703-713, May 1999.

[5] 우균, “공간 조정을 이용한 복사 메모리 재사용 알고리즘,” *동아대학교 부설 정보기술연구소 논문집*, 제9권 제1호, pages 131-139, 2001년 8월.

[6] 우균, 김성청, 주성용, 차미양, 한태숙, “공간 조정을 이용한 복사 메모리 재사용 알고리즘,” *한국정보과학회 제28회 추계학술대회 논문집*, Vol.28, No.2, pages 346-348, 2001.10.

[7] R. Bird, *Introduction to Functional Programming using haskell*, 2nd Ed., Prentice hall Europe, 1998.

[8] S. B. Lippman and J. Lajoie, *C++ Primer*, 3rd Ed., Addison-wesley, 1997.

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

[10] S. L. Peyton Jones and D. R. Lester, *Implementing Functional Language: a tutorial*, Prentice Hall, 1991.

[11] T. Johnsson, *Compiling Lazy Functional Language*, PhD thesis, Chalmers Tekniska Högskola, Göteborg, Sweden, January 1987.

[12] W. Partain, “The nofib benchmark suite of Haskell programs,” In J. Launchbury and P. M. Samson, editors, *Functional Programming, Glasgow*, Workshops in computing, pages 195-202, Springer Verlag, 1992.

김성청
1995년~1997년 동부산대학 사무자동화과
1997년~2000년 한국방송대학교 컴퓨터과학과(학사)
200년~현재 동아대학교 컴퓨터 공학과 석사과정 재학중
관심분야는 지연 함수형 언어의 구현, 메모리 재사용 시스템 등

우 균
1987년~1991년 한국과학기술원 전산학과(학사)
1991년~1993년 한국과학기술원 전산학과(석사)
1998년~2000년 한국과학기술원 전산학과(박사)
2000년~현재 동아대학교 컴퓨터공학과 전임강사
관심분야는 함수형 언어 프로그래밍 환경, 지연 함수형 언어를 위한 추상기계, 프로그램의 정적 분석 방법 등