논·문15-1-2

ZG-machine을 위한 비재귀적 메모리 재사용†
(A Non-recursive Garbage Collection for the ZG-machine)

우 균
동아대학교 전기전자컴퓨터공학부 컴퓨터공학전공
woogyun@daunet.donga.ac.kr

요 약

ZG-machine은 태그 옮김이라는 간단한 부호화 기법을 통해 공간 효율을 높인 G-machine으로서 G-machine과 비교하여 힙 공간을 평균 30% 절약할 수 있었다. 그러나 ZG-machine은 G-machine에 비해 수행시간 부담이 다소 증가하였는데, 수행시간의 증가는 주어진 힙 크기에 따라 다르게 나타났다. 실험 결과를 분석한 결과 이 실행시간 증가는 ZG-machine의 메모리 재사용 체계의 비효율성 때문인 것으로 판단되었다. 현재 ZG-machine의 메모리 재사용 체계는 비효율적인 재귀적 알고리즘을 사용하고 있는데, 그 이유는 태그 옮김 기법에 따른 노드 구조 해체로 인해 비재귀적 알고리즘을 직접 적용하기 힘들기 때문이다. 본 논문에서는 ZG-machine을 위한 비재귀적 메모리 재사용 알고리즘을 제안하고, 이 알고리즘의 성능을 실험을 통해 확인한다. 비재귀적 메모리 재사용 알고리즘은 재귀 호출에 필요한 별도의 스택공간을 사용하지 않는다는 점에서 더 경제적이고 함수 호출에 걸리는 시간을 줄일 수 있다는 면에서 더 빠를 것으로 기대되었는데, 실험 결과에 따르면 비재귀적 메모리 재사용 알고리즘은 다량의 힙을 소모하는 프로그램에 대해서만 제한적으로 속도 향상을 보이는 것으로 나타났다.

1. 서론

ZG-machine은 태그 옮김이라는 부호화 기법을 통해 공간 효율을 개선한 G-machine이다[1, 2, 3]. G-machine[4, 5]은 지연 함수형 언어를 구현하기 위한 추상기계로 아직 계산되지 않은 표현식을 나타내기 위해 그래프를 사용한다. G-machine의 그래프 구조는 하나의 태그와 하나 이상의 데이터 필드로 이루어져 있는데, ZG-machine은 G-machine 그래프 노드의 태그를 해당 노드를 가리키는 포인터 방향으로 옮겨 포인터와 함께 저장함으로써 힙 공간을 옮겨진 태그 크기만큼 절약한다.

† 이 논문은 2000학년도 동아대학교 학술연구조성비(신진과제)에 의하여 연구되었음.

최근 실험 결과[3, 6]에 의하면, ZG-machine의 수행속도는 G-machine에 비해 다소 증가되었는데, 그 이유는 ZG-machine의 메모리 재사용 시스템의 비효율성 때문인 것으로 추정된다. ZG-machine은 G-machine에 비하여 힙 공간을 평균 30% 절약할 수 있지만 수행 속도는 다소 증가한 것으로 나타났는데 구체적인 증가율은 주어진 힙 크기에 따라 다르게 나타났다. 이는 힙 크기에 따른 변화, 즉 메모리 재사용에 따른 변화에 수행 속도가 종속적인 것을 의미한다. ZG-machine의 수행속도는 특히 힙 공간의 제약이 최대일 때—프로그램 수행에 필요한 최소한의 힙이 주어졌을 때 현저하게 속도가 감소하였는데, 이는 메모리 재사용 시스템의 호출이 증가했기 때문인 것으로 추정된다.

ZG-machine은 G-machine과 달리 보다 효율적인 비재귀적 메모리 재사용(non-recursive garbage collector) 알고리즘[7]을 사용할 수 없고 재귀적 메모리 재사용(recursive garbage collector) 알고리즘[8]을 사용할 수밖에 없었는데, 그 이유는 태그 옮김에 따라 노드 구조가 해체되어, 비재귀적 알고리즘을 직접 적용할 수 없었기 때문이다. 비재귀적 알고리즘은 그래프를 너비 우선 방식으로 탐색하는 방식이고, 재귀적 알고리즘은 그래프를 깊이 우선 방식으로 탐색하는 알고리즘이다. 재귀적 알고리즘은 재귀 호출에 따른 수행시간 부담과 공간 부담이 있기 때문에 ZG-machine의 실용성을 높이기 위해 반드시 극복해야 할 과제이다.

본 논문은 ZG-machine을 위한 비재귀적 메모리 재사용 알고리즘을 소개하고 간단한 실험을 통해 제안된 비재귀적 메모리 재사용 알고리즘의 성능을 살펴본다. ZG-machine을 위한 비재귀적 메모리 재사용 알고리즘도 기본적으로 Cheney 알고리즘[7]에 기반하고 있다. 다만 옮겨진 태그를 처리하기 위해 약간의 수정이 필요하다.

본 논문의 구조는 다음과 같다. 먼저 2절에서는 ZG-machine에서 사용하고 있는 태그 옮김 기법을 소개한다. 3절에서는 ZG-machine을 위한 메모리 재사용 시스템에 비재귀적 알고리즘을 적용하는 것이 힘든 이유를 설명한다. 4절에서는 ZG-machine을 위한 비재귀적 메모리 재사용 알고리즘을 제안하고 이 알고리즘의 성능에 대해 살펴본 후, 마지막으로 5장에서 결론을 맺는다.

2. 태그 옮김 기법

태그 옮김 기법은 G-machine의 노드 종류가 그렇게 다양하지 않다는 것에 착안한 방법이다. G-machine의 노드 종류는 몇 개에 지나지 않지만 현재 G-machine의 구현은 이 태그를 하나의 워드에 저장하도록 하고 있다. 그 이유는 태그에 따라 다른 수행코드를 수행하고자 할 때, 이러한 코드의 시작 주소들이 저장되어 있는 테이블의 주소를 태그로 함으로써 태그를 해석하는 부담을 줄이기 위해서이다[9, 10].

그러나 태그를 한 워드에 저장하는 것은 심각한 공간 낭비로 생각되며, 이러한 공간 낭비는 실행 시간에 메모리 참조 횟수를 높이게 되므로 수행 시간의 측면에서도 그렇게 큰 이점이 없을 것으로 생각된다. 그런데 태그를 힙 워드 하나에 저장하는 대신 몇 비트를 사용하여 작은 공간에 저장한다고 하여도, 이로 인해 절약된 공간을 활용할 방법이 특별히 없기 때문에 현재까지의 G-machine

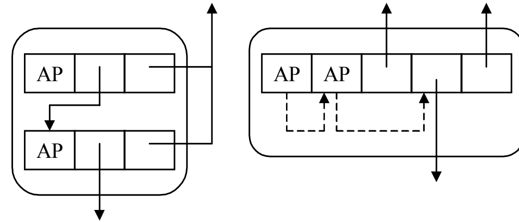
은 태그를 그냥 한 워드에 저장하도록 하고 있는 것으로 생각된다.

ZG-machine이 채택하고 있는 태그 옮김 기법은 이 남는 공간에 상대주소를 저장하도록 함으로써 남는 공간을 활용한다. G-machine의 그래프 구조는 그래프 노드가 여러 절대주소 포인터에 의해 얽혀 있는 구조이다. 만약 이러한 절대주소 포인터 중 일부를 상대주소로 표현할 수 있다면 이를 태그와 함께 저장함으로써 힙 공간을 절약할 수 있다.

G-machine의 그래프 중 일부는 실제로 컴파일 시간에 그 형태가 결정되는 부분이 있다. 이는, 해당 부분의 그래프 노드의 실제 주소를 알 수는 없어도, 상대적인 위치는 컴파일 시간에 파악할 수 있음을 의미한다. 따라서 이를 활용하여 태그 옮김을 할 수 있다.

그림 1은 ZG-machine의 태그 옮김 기법을 설명하고 있다. 그림 1의 (a)는 G-machine의 그래프 구조 일부를 나타낸다. 만약 여기서 일부분이 동시에 힙에 할당된다는 것을 알 수 있다면, 이 부분 내부의 포인터는 상대주소로 변환될 수 있고 이 상대주소는 그 포인터가 가리키는 노드의 태그와 함께 저장될 수 있다. 이를 위해서 노드의 태그는 해당 포인터의 위치로 이동된다. 그림 1의 (b)는 (a)의 그래프에 대하여 태그 옮김이 일어난 그래프를 나타낸다. 여기서 실선 화살표는 절대주소를 점선 화살표는 상대주소를 나타낸다.

그림 1의 (b)에서는 태그 옮김이 일어나지 않은 노드인 첫 번째 AP 노드에 대해서도 상대주소가 부여되어 있음을 볼 수 있는데, 이는 태그 옮김이 일어난 후의 그래프에서도 태그를 같은 방법으로 취급하기 위해서이다. 또한 그림 1의 (b)는 절대주소와 상대주소가 섞여있는 것을 볼 수 있는데, 이 두 가지 주소를 구분하기 위해서 별도의 플래그가 필요하다.



(그림 1) 태그 옮김 기법의 예

결과적으로 태그 옮김이 일어난 후의 그래프는 (옮겨진 태그 수)×(힙 워드의 크기) 만큼의 힙 공간을 절약할 수 있다. 그림 1의 경우에는 단 하나의 힙 워드만 절약되었지만 (1/6 = 17% 절약) 실험 결과에 따르면 ZG-machine은 태그 옮김을 통해 평균 30% 가량의 힙 공간을 절약할 수 있었다.

3. 기존 ZG-machine의 메모리 재사용

현재 ZG-machine은 재귀적 메모리 재사용 시스템을 채택하고 있다. 메모리 재사용 알고리즘에는 여러 가지가 있지만, 복사에 의한 메모리 재사용 알고리즘은 그 단순성으로 인해 널리 사용되고 있다. 복사에 의한 메모리 재사용 알고리즘은 크게 재귀적 알고리즘과 비재귀적 알고리즘으로 구분되는데, 비재귀적 알고리즘이 일반적으로 더 효율적이지만, ZG-machine은 비재귀적 알고리즘을 직접 사용하는데 문제가 있다.

ZG-machine에 비재귀적 메모리 재사용 알고리즘을 직접 적용하기 어렵다는 사실을 살펴보기 전에 먼저 비재귀적 복사 알고리즘을 살펴보자. Cheney에 의한 비재귀적 메모리 재사용 알고리즘을 C 언어와 유사한 형태의 의사코드로 기술하면 그림 2와 같다. 그림 2의 의사코드는 Jones와 Lins가 정리한 내용[11]을 따랐다.

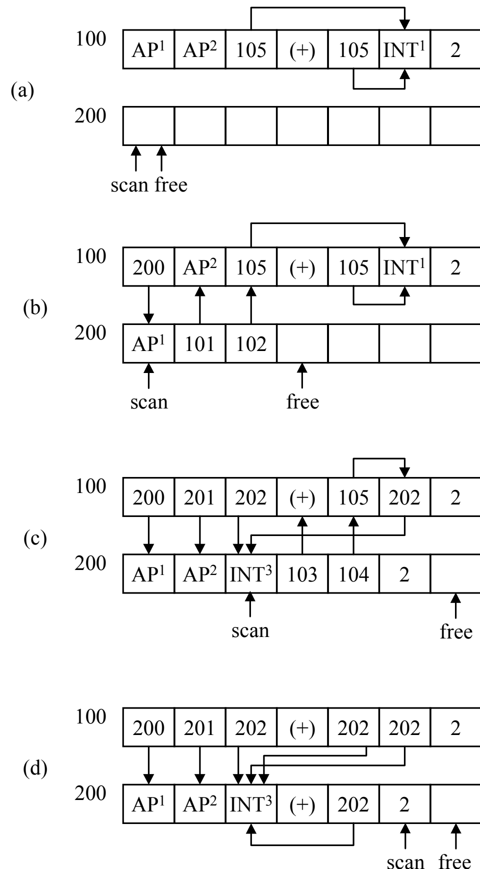
```
flip() =
    swap(FromSpace, ToSpace);
    free = scan = ToSpace;
    for root in roots
        root = copy(root);
    while scan < free
        for p in children(scan)
        *p = copy(*p);
        scan = scan + size(scan);

copy(p) =
    if forwarded(p) then
        return forwardAddress(p);
    else
        addr = free;
        move(p, free);
        free += size(p);
        forwardAddress(p) = addr;
        return addr;
```

(그림 2) Cheney 알고리즘

Cheney 알고리즘은 모든 유효그래프의 루트를 먼저 새로운 공간으로 복사한다. 이는 프로시저 flip의 for 문에서 이루어진다. 그리고 복사된 노드가 원래 있던 자리에는 해당 노드가 옮겨진 위치, 즉 새로운 공간상의 주소를 기입해 둔다. 이 주소를 전방주소(forward address)라고 한다. 만일 이미 이동된 노드를 재차 복사하려는 시도가 일어날 경우에는, 전방주소를 통해 이동되었다는 사실을 알 수 있고, 동시에 실제 이동된 주소도 알 수 있으므로 그래프의 연결 구조를 원래대로 재구성할 수 있다.

이 알고리즘을 그대로 ZG-machine에 직접 적용해 보면 문제점이 발생하는데, 이를 보기 위해 그림 3을 보자. 그림 3은 Cheney 알고리즘을 아무 수정 없이 ZG-machine에 적용하였을 때 발생하는 문제점을 나타낸다.



(그림 3) ZG-machine에서 Cheney 알고리즘이 실패하는 예

그림 3은 100번지부터의 그래프를 200번지에서 시작하는 새로운 공간으로 복사하는 과정을 나타낸다. 그림 3에서 태그에 붙은 첨자는 상대주소를 나타내고 숫자는 실제로 해당 워드에 저장되어 있는 수치를 나타내며, (+)는 해당 함수 노드의 주소를 나타낸

다. 여기서 함수 노드는 메모리 재사용 시 수거하지 않는 것으로 가정하였다. 그림 3의 유효 그래프 루트는 100번지에 존재하는 것 하나 뿐이다. ZG-machine의 그래프에 있어서, 비록 태그가 옮겨지긴 했지만 각 태그의 상대주소를 이용하여 노드 데이터 필드를 참조할 수 있기 때문에 데이터 필드를 복사하는 것은 G-machine과 똑같은 방법으로 수행될 수 있다.

그림 3(a)는 메모리 재사용을 위한 복사가 일어나기 전 초기 상태를 나타낸다. 초기 상태에서 scan 포인터와 free 포인터는 같은 지점, 즉 옮겨질 새로운 공간을 가리키고 있다. 여기서 루트 노드 하나가 복사되면 그림 3(b)와 같은 상태가 된다. 일단 노드가 복사되면 노드가 원래 있던 자리에는 전방주소가 기입된다. 여기서 전방주소는 200이다. 노드 태그가 복사될 때 상대주소도 다시 계산된다는 것에 주의하자. 그림 3(b)의 AP 태그 상대주소는 1로 변화가 없는 것처럼 보이지만지만 사실 이 1은 복사가 일어난 후 다시 계산된 상대주소이다.

그림 3(c)는 그림 3(b)에서 scan 포인터를 두 번 움직여 루트가 가리키던 노드를 두 개 복사한 후의 상황을 나타낸다. 이 과정에서도 마찬가지로 해당 노드의 태그가 이동되고 상대주소가 다시 계산된다.

ZG-machine에서 복사 알고리즘의 문제는 가장 마지막 단계인 그림 3(d)에서 볼 수 있다. 그림 3(d)는 그림 3(c)에서 추가로 두 개의 노드에 대한 검사를 마친 후의 단계를 나타내는데, 이 단계에서 scan 포인터가 주소 데이터가 아닌 보통의 정수(일반적으로 기타 기본 자료형의 자료)를 가리키고 있는 것을 볼 수 있다. 이 때, 해당 INT 노드의 태그가 분리되어 앞쪽에 있어서 태그를 참조할 수 없기 때문에, scan 포인터는 이 자료가 정수라는 것을 알 수 없다. 따라서 scan 포인터가 접하는 모든 노드(데이터 필드로만 이루어져 있는 힙 워드)의 자료 종류에 대하여 어떤 가정도 할 수 없는 상황이 발생한다. 지금까지는 태그가 아닌 모든 워드는 절대주소를 가리킨다고 가정하고 여기까지 왔지만, 결국 이러한 가정은 일반적으로 할 수 없으며 잘못된 가정이었다는 말이 된다. 결과적으로, Cheney 알고리즘을 ZG-machine에 직접 적용할 수 없다. 이를 해결하는 방법을 다음 절에서 알아보자.

4. ZG-machine을 위한 비재귀적 메모리 재사용 알고리즘

앞 절에서 우리는, 태그가 분리되어 이동되었다는 것 때문에 ZG-machine에 비재귀적 메모리 재사용 알고리즘을 직접 적용할 수 없음을 알았다. 이 절에서는 ZG-machine을 위한 비재귀적 메모리 재사용 알고리즘을 살펴본다.

4.1 기본 아이디어

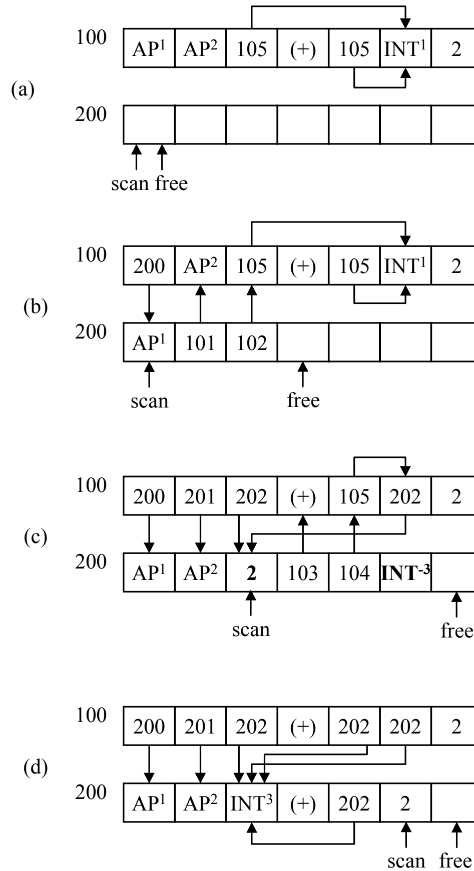
Cheney 알고리즘을 아무 수정없이 ZG-machine에 적용하면, scan 포인터를 통해 그래프를 검사할 때 문제가 발생한다. scan 포인터를 통해 이미 이동된 노드를 검사할 때, ZG-machine의 경우에는 노드 데이터 필드만을 검색하게 되는데, 그 이유는 노드 태그가 이미 전방으로 이동되었기 때문이다. 따라서 노드 데이터 필드가 주소인지, 기본 자

료형의 자료인지 판단할 수 없다.

이는 scan 포인터에 의한 노드 검사 이전에 노드 자료 복사 방법을 조금 변경함으로써 해결할 수 있다. 어떤 한 노드를 복사하는 시점에서 우리는 해당 태그를 알 수 있다. 따라서 이 때, 해당 노드의 종류를 알 수 있도록 노드 정보를 저장해 둔다면 나중에 scan 포인터를 통해 검사할 때, 이 정보를 통해 해당 노드의 종류를 알 수 있다. 문제는 이 정보를 저장할 공간이 특별히 마련되어 있지 않다는 것인데, 이는 태그가 저장되는 위치와 자료가 저장되는 위치를 바꿈으로써 해결할 수 있다. 구체적인 예를 위해 그림 4를 보자.

그림 4의 (a)와 (b)는 그림 3의 경우와 같다. 그림 4에서 주목할 점은 (c)의 경우, 즉 INT 노드를 복사하는 경우이다. 그림 4에서는 INT 노드를 복사할 때, 데이터 필드를 그림 3에서 저장했던 원래 Cheney 알고리즘에 따른 위치에 저장하지 않고 태그가 저장되어야 할 위치인 202번지에 저장하였다. 대신 원래 데이터 필드가 저장되어야 하는 위치인 205번지에는 해당 INT 태그와 상대주소를 저장해 둔다. 간단히 말해서 기본 자료형에 대해서는 자료가 저장되는 위치와 태그가 저장되는 위치를 바꾸어 저장하는 것이다. 바뀌어 저장된 태그와 자료는 그림 4(c)에 굵은 글꼴로 나타내었다.

기본 자료형에 대하여 태그와 자료를 바꾸어 저장할 때 주의할 점은 해당 태그의 상대주소를 음수로 저장한다는 것이다. 그리 4(c)의 205번지에 저장된 INT 태그를 보면 상대주소가 -3으로 저장되어 있는 것을 알 수 있다. 그래프 생성 시 태그 옮김이 일어날 때에는 항상 해당 상대주소가 양의 정수가 되도록 할 수 있다. 따라서 상대주소가 음수라는 말은 메모리 재사용을 위한 검사가 진행중이고 이미 해당 노드를 한 번 검사했다는 것을 나타내는 것이다. 이는 결과적으로 상대주소에 한 비트의 플래그를 더 두는 것과 실제적으로 같은 효과이다.



(그림 4) 비재귀적 메모리 재사용 알고리즘의 동작 예

이제 scan 포인터가 데이터 필드를 검사할 때는 모든 자료가 주소라고 가정할 수 있다. ZG-machine에서 주소 자료는 절대주소와 상대주소로 구분되고 상대주소는 항상 태그와 함께 존재한다. 그러나 이들 두 주소 워드를 구분하는 플래그가 존재하므로, 태그와 절대주소를 구분하는 것은 문제가 되지

않는다. 대신 scan 포인터는 태그와 상대주소로 이루어진 힙 워드를 만났을 경우 상대주소가 음수인 경우에 해당 자료와 태그를 바꾸어 줌으로써 원래의 그래프 형태를 유지해야 한다. 그림 4(d)는 기본 자료형의 태그에 대하여 이러한 교환 과정이 이루어지는 것을 나타내고 있다.

이 교환과정에서 음수로 저장해 두었던 태그의 상대주소가 이용된다. 원래 태그가 저장되어야 할 위치는 현재 태그가 저장되어 있는 위치에서 해당 상대주소의 절대값 만큼 뒤로 이동한 위치이다. 즉, 위 그림 4(c)에서 원래 INT 태그가 저장되어야 할 주소인 202번지는 현재 INT 태그가 저장되어 있는 205번지에서 상대주소 -3만큼을 더한 위치가 된다.

Cheney 알고리즘에서 모든 노드는 한 번씩만 검사되므로, 즉 scan 포인터는 모든 노드를 한 번씩만 거치게 되고 노드 교환 이후에 다시 그 노드를 검사하게 되는 경우는 발생하지 않는다. 따라서 그림 4의 방법은 제대로 작동된다.

4.2 비재귀적 알고리즘의 성능

이 절에서는 제안된 비재귀적 메모리 재사용 알고리즘의 성능을 살펴본다. 먼저 알고리즘 자체의 성능을 정성적으로 고찰해 본 후 실험을 통해 정량적인 결과를 살펴본다.

개선된 비재귀적 메모리 재사용 방법의 효율을 정성적으로 평가해 보자. 이 방법은 원래의 Cheney 알고리즘에 비하여 scan 포인터로 검사할 때 기본 자료형 노드에 대해서는 해당 자료와 태그를 바꾸어야 한다는 부담이 있다. 또 이를 위해서 상대주소가 음수 값인가 검사하는 작업이 필요하다. 따라서 여전히 원래의 Cheney 알고리즘보다는 수행시간이 길어질 것이다. 그러나, 재귀적 재사용 알고리즘에서 필요했던 함수 재귀 호출로 인한 시간 부담에 비하여 상당히 적을 것으로 생각된다. 또한 재귀 호출로 인해 추가로 필요했던 스택 공간은 더 이상 필요하지 않다는 장점이 있다.

(unit: 10ms)

		GC 횟수	평균 GC 시간	비율
exp	ZGM	789	1.414829	100%
	ZGM+	827	0.754171	53%
	GM	1179	0.506361	36%
nfib	ZGM	887	0.018151	100%
	ZGM+	889	0.025084	138%
	GM	1261	0.012688	70%
primes	ZGM	13	0.630770	100%
	ZGM+	14	0.678572	108%
	GM	19	0.484212	77%
queens	ZGM	829	0.600000	100%
	ZGM+	874	0.633066	106%
	GM	1226	0.443475	74%
tak	ZGM	1052	0.075856	100%
	ZGM+	1060	0.087547	115%
	GM	1527	0.040799	54%

(표 1) 비재귀적 알고리즘의 성능

제안된 알고리즘의 성능을 정량적으로 평가하기 위해, 제안된 알고리즘을 구현하여 간단한 실험을 수행해 보았다. 표 1은 nofib 벤치마크(Imaginary)[12]의 이매지너리(Imaginary) 프로그램에 대한 각 기계의 GC 성능을 측정한 결과를 보여준다. 실험 환경으로는 2GB 메모리를 갖춘 Sun Ultra Sparc 웍스테이션을 사용하였으며, 컴파일러

는 GNU C 컴파일러 버전 2.8.1을 사용하였고 컴파일 옵션은 -O 옵션을 선택하였다.

표 1에서 ZGM은 재귀적 알고리즘을 사용한 ZG-machine을 나타내고, ZGM+는 비재귀적 알고리즘(변형된 Cheney 알고리즘)을 사용한 ZG-machine을 나타낸다. GM은 비재귀적 알고리즘(Cheney 알고리즘)을 사용한 G-machine을 나타낸다. 각 경우에 대하여 힙 크기를 1M로 제한하였으며 메모리 재사용 시간은 10miliscond 단위로 측정하였다. 메모리 재사용 시스템 호출 횟수를 측정하여 평균 메모리 재사용 시스템 수행시간을 산출하였으며, 각 프로그램을 10회 수행한 결과의 평균으로 메모리 재사용 시스템 수행시간을 결정하였다.

표 1은 매우 흥미로운 결과를 보여준다. 먼저 GC 횟수를 살펴보면, ZGM+의 메모리 재사용 횟수가 ZGM보다 더 많다는 것을 알 수 있다. 이는 본 논문에서 제안한 비재귀적 메모리 재사용 알고리즘이 기존의 재귀적 알고리즘보다 더 적은 힙 공간을 확보한다는 말이 된다. 이러한 결과는 G-machine의 경우에는 발생할 수 없지만, ZG-machine의 경우에는 메모리 재사용 알고리즘 수행시 태그 옮김에 따른 그래프 압축이 추가적으로 일어나기 때문에 가능하다.

ZGM+의 GC 횟수가 ZGM보다 많다는 것은 바꾸어 말하면 ZGM+의 경우 그래프 크기가 ZGM보다 크다는 것을 의미한다. 이는 즉 ZGM+의 메모리 재사용 체계가 더 큰 그래프를 옮겨야 한다는 것을 의미하고, 따라서 메모리 재사용 체계가 느려질 수 있다는 것을 의미한다.

표 1의 가장 우측 열에 보인 비율은 ZGM에 대한 ZGM+와 GM의 평균 메모리 재사용 시간을 나타낸다. 실험 결과에 따르면 ZG-machine을 위한 비재귀적 알고리즘은 재귀적 알고리즘보다 대부분 낮은 효율을 보이고 있고, exp의 경우에만 높은 효율을 보이고 있음을 알 수 있다. 기존 실험[3,6]에 따르면, nofib 벤치마크의 이매지너리 프로그램 중에서 exp의 경우에 ZG-machine의 성능이 G-machine에 비해 가장 저조한 것으로 나타났는데, 이는 곧 프로그램 exp가 상대적으로 많은 양의 힙을 소모하는 프로그램임을 의미한다. 결국 ZGM+는 다량의 힙을 소모하는 프로그램에 대해서는 기존 ZGM보다 높은 성능을 보인다고 추정할 수 있다. 표 1에 따르면 프로그램 exp에 대해서 ZGM+의 메모리 재사용 체계는 ZGM에 비해 53%의 시간만을 필요로 한다.

5. 결론

이상에서 ZG-machine을 위한 비재귀적 메모리 재사용 알고리즘을 제안하였고 제안된 비재귀적 알고리즘의 성능을 간단한 실험을 통해 알아보았다. 제안된 비재귀적 메모리 재사용 알고리즘은 그래프를 깊이 우선 방식으로 탐색하기 위한 재귀 호출이 없다는 점에서 재귀적 알고리즘에 비해 스택 공간을 절약할 수 있다. 그러나 시간 효율이 항상 향상된 것은 아니었는데, 메모리 사용이 방대한 프로그램에 대해서만 시간 효율이 향상된 것을 알 수 있었다.

이 논문에서 제안한 알고리즘은 여전히 G-machine에서 사용하는 원래의 Cheney 알고리즘보다 비효율적인 것으로 드러났다. 본 논문에서 제안한 방법은 기본적으로 상대주소를 다루는 작업이 필요하고, 기본 자료형

에 해당하는 노드에 대하여 태그와 해당 자료를 교환하는 작업이 필요하다. 이는 ZG-machine의 태그 옮김 특성으로 인해 어쩔 수 없이 가중되는 부담인 것으로 생각된다.

실험 결과에 따르면 제안된 방법이 기존의 재귀적 방법보다 메모리 재사용시 그래프 압축 비율이 낮아지는 것으로 판명되었는데, 이는 제안된 알고리즘이 압축 비율 면에서 개선될 여지가 있음을 의미한다. 압축 비율 개선은 메모리 재사용 체계의 속도에도 기여할 것으로 예측된다.

참고문헌

[1] Gyun Woo and Taisook Han, "Compressing the Graphs in the G-machine by Tag-Forwarding," *Journal of Computing and Information*, 3(1):112-138, 1998.

[2] 우균, 한태숙, "태그 옮기기에 의한 G-machine 그래프의 압축," *정보과학회 논문지(B)*, 26(5):702-712, 1999년 5월.

[3] 우균, *태그 옮김 기법으로 공간 효율을 높인 G-machine*, 박사학위논문, 한국과학기술원, 2000년 2월.

[4] L. Augustsson, "A Compiler for Lazy ML," In *Proceedings of the 1986 ACM Conference on Lisp and Functional Programming*, pages 218-227, August 1984.

[5] T. Johnsson, "Efficient compilation and lazy evaluation," In *Proceedings of the ACM SIGPLAN '84 Symposium on Compiler Construction*, pages 58-69, June 1984.

[6] 우균, 한태숙, "ZG-machine에서 기억장소 재활용 체계의 영향," *정보과학회 논문지: 소프트웨어 및 응용*, 27(7):759-768, 2000년 7월.

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

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

[9] Thomas Johnsson, *Compiling Lazy Functional Languages*, PhD Thesis, Chalmers Tekniska Högskola, Göteborg, Sweden, January, 1987.

[10] S. L. Peyton Jones, *The Implementation of Functional Programming Languages*, Prentice Hall, 1987.

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

[12] W. Partain, "The nofib Benchmark Suit of Haskell Programs," In J. Launchbury and P. M. Samson, editors, *Functional Programming, Glasgow,* Workshop in Computing, pages 195-202, Springer Verlang, 1992.

우 균

1987년~1991년 한국과학기술원 전산학(학사).

1991년~1993년 한국과학기술원 전산학(석사).

1993년~2000년 한국과학기술원 전산학(박사).

2000년~현재 동아대학교 전기전자컴퓨터공학부 전임강사.

관심분야는 지연 함수형 언어의 구현과 이를 위한 추상 기계, 프로그램 변환 등임.