# 메모리 재사용 명령어 자동 삽입 변환기의 효과*

이욱세○ 이광근
서울대학교 컴퓨터공학부
{oukseh; kwang}@ropas.snu.ac.kr

# Experiments on the Effectiveness of an Automatic Insertion of Safe Memory Reuses into ML-like Programs

Oukseh Lee○ Kwangkeun Yi
School of Computer Science & Engineering, Seoul National University

요 약

ML 프로그램에 메모리 재사용 명령어를 자동으로 삽입하는 변환기의 효과에 대한 실험 결과를 보인다. 분석 및 변환에 드는 비용은 초당 1,582 줄-29,000 줄이었다. 총 생성 메모리의 3.8%-88.6%를 재사용하도록 변환함으로써 메모리 최고점(memory peak)을 0.0%-71.9% 감소시켰다. 재사용에 의한 프로그램 실행 속도는 25.4% 단축되거나 42.9% 느려졌다. 프로그램 실행 시간 중에 메모리 수거(garbage collection)의 비중이 높을 경우에만 수행 속도가 단축되었다.

## 1 서론

ML 프로그램[1]에 메모리 재사용 명령어를 자동으로 삽입하는 변환기[2]는 메모리를 많이 재사용하도록 프로그램을 변환하나, 재사용에 따른 시간적·공간적 이익이 얼마나 있는지는 알려지지 않았다.

이 논문은 다음 질문에 대한 답을 제공한다. 메모리 재사용 변환에 의해 메모리 최고점(memory peak)이나 메모리 수거(garbage collection, 이하 GC) [3] 성능에 미치는 영향은 무엇인가? 메모리 재사용 변환에 의해 프로그램의 전체 실행 시간은 어떻게 변하는가?

## 2 변환기

본 논문의 실험 대상인 변환기[2]는 ML 프로그램에 메모리 재사용 명령어를 삽입하여 프로그램이 무조건 메모리를 요구하는 것을 막는다. 실제로는 메모리 반납 (deallocation) 명령어를 삽입하는데, 메모리 생성 (allocation) 직전에 삽입하므로 메모리 재사용과 같은 효과를 가진다.

**예제 1.** 다음 `insert` 함수는 정렬된 입력 리스트 `l`에 정수 `i`를 삽입하여 정렬된 리스트를 준다.

```
fun insert i l =
  case l of
    []    => i::[]                          (1)
  | h::t => if i<h then i::l                 (2)
            else let z = insert i t          (3)
                 in  h::z                    (4)
```

입력 리스트 `l`이 함수 호출 이후에 사용되지 않는 가정 아래 C 언어로 같은 함수를 작성한다면, `i`를 위한 메모리 셀 (cell) 하나만 생성하고 입력 리스트의 내용을 고쳐서 셀을 삽입해 줄 것이다. 그러나, 위 ML 프로그램의 `(4)` 줄에서는 재귀 호출의 회수 만큼 새로운 셀을 생성한다.

본 변환기는 프로그램의 메모리 사용 행태에 대해 분석을 한 후, 분석 결과를 토대로 `insert` 함수를 다음과 같이 변환한다.

```
fun insert b i l =
  case l of
    []    => i::[]                          (1)
  | h::t => if i<h then i::l                 (2)
            else let z = insert b i t        (3)
                 in  (free l when b; h::z)   (4)
```

추가의 인자 `b`를 받는데, `b`가 참이면 입력 리스트가 함수 호출 이후에 쓰이지 않는다는 뜻이다. `(4)` 줄의 "`free l when b`"은 `b`가 참일때 `l`이 가리키고 있는 메모리 셀을 반납한다. 반납 직후에 메모리 셀을 생성하므로 두 명령어를 묶어 재사용으로 구현 가능하다.

변환된 함수를 "`insert true 5 l`"와 같이 호출하면 매번 재귀호출 때 새로운 셀을 생성하지 않으므로 5를 위한 셀 하나만 생성하게 된다. □

본 변환기는 일반적인 ML 프로그램에 대해 적용 가능하다. 리스트 뿐만 아니라 나무 구조 (tree), 그래프 (graph) 등에 적용 가능하나, ML 프로그램에서 잘 사용되지 않는 사이클(cycle)이 있는 구조에는 적용 불가하다. 자세한 알고리즘은 [2]에 있다.

## 3 분석 및 변환 비용과 메모리 재사용 비율

분석 및 변환 비용은 Pentium4에서 초당 1,582-29,000 줄이었다 (그림 1.(a)의 A 열). 그림 1의 그래프 (c)는 분석·변환기의 복잡도가 이론적으로 기하급수적 임에도 불구하고 실제로는 프로그램 크기의 제곱보다 작음을 보여준다.

변환에 의해 총 생성 메모리의 3.8%-88.6%가 재사용되었다. (그림 1.(a)의 C 열). 재사용 비율이 낮은 queens와 kb의 경우 프로그램 실행 중에 그래프 형태의 자료 구조를 많이 사용한다. 예를 들어, kb는 단어 치환 함수를 많이 사용하는데 치환된 결과는 일부 단어를 공유하는 그래프 형태이다. 이러한 경우 본 분석 및 변환기는 재사용 명령어를 많이 삽입하지 못하였다.

## 4 메모리 최고점 (Memory Peak)

변환에 의해 메모리 최고점이 0.0%-71.9% 낮아졌고 (그림 1.(b)) 이는 메모리 재사용 비율과 상관 관계가 있다 (그

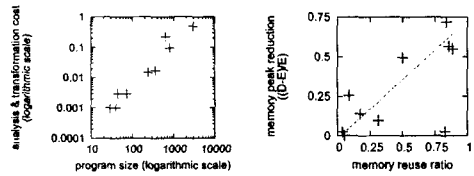
*본 연구는 두뇌한국 21의 지원을 받았음을 밝힙니다.

| 프로그램 | 줄 | A 변환 | B 할당 | C 재사용 | C/B |
|---|---|---|---|---|---|
| sieve | 29 | 0.001 | 30829 | 26423 | 85.7% |
| merge | 40 | 0.001 | 5860 | 2930 | 50.0% |
| qsort | 41 | 0.001 | 35997 | 30148 | 83.7% |
| queens | 44 | 0.003 | 34641 | 1807 | 5.2% |
| msort | 73 | 0.003 | 21506 | 19064 | 88.6% |
| mirage | 245 | 0.015 | 20381 | 16857 | 84.4% |
| life | 366 | 0.017 | 10036 | 875 | 8.7% |
| k-eval | 645 | 0.220 | 52894 | 16684 | 31.5% |
| kb | 808 | 0.095 | 24473 | 940 | 3.8% |
| nucleic | 3019 | 0.488 | 31092 | 5491 | 17.7% |

(a) 분석 및 변환 시간, 메모리 재사용 비율.

| 프로그램 | 줄 | D 최고점 | E 재사용시 최고점 | (D-E)/E |
|---|---|---|---|---|
| sieve | 29 | 690 | 300 | 56.5% |
| merge | 40 | 1197 | 606 | 49.4% |
| qsort | 41 | 1189 | 334 | 71.9% |
| queens | 44 | 255 | 255 | 0.0% |
| msort | 73 | 714 | 321 | 55.0% |
| mirage | 245 | 1398 | 1361 | 2.6% |
| life | 366 | 2346 | 1746 | 25.6% |
| k-eval | 645 | 1044 | 944 | 9.6% |
| kb | 808 | 27125 | 26501 | 2.3% |
| nucleic | 3019 | 103677 | 89352 | 13.8% |

(b) 메모리 최고점 감소.




(c) 변환 비용 (경사 1.46) (d) 최고점 하락과 재사용율의 관계

A 초: 분석 및 변환 시간 (Objective Caml 3.06 컴파일러[4], Intel Pentium4 3.0C, Linux RedHat 9.0).
B kilo words: 프로그램 실행 중에 할당된 메모리 셀 수.
C kilo words: 삽입된 명령어에 의해 재사용된 셀 수.
D words: 변환전 프로그램의 실행 중 메모리 최고점.
E words: 변환된 프로그램의 실행 중 메모리 최고점.

그림 1: 분석 및 변환 비용, 재사용 비율, 메모리 최고점.

림 1.(d)). 메모리 최고점은 프로그램 실행 도중에 살아 있는, 즉, 스택(stack)으로부터 도달 가능한 메모리 셀들의 수의 최대값이다. sieve, merge, qsort, msort의 경우 재사용 비율도 높고 최고점 하락도 크다. queens와 kb의 경우 재사용 비율이 낮고 최고점 하락도 작다. 그러나, life와 mirage의 경우 재사용 비율과 최고점 하락이 일치하지 않는다. mirage의 경우 재사용 비율은 높고 (84.4%) 반면에 최고점 하락은 작다 (2.6%). 그림 2를 보면 뒤쪽 부분의 최고점 부분에서 재사용이 거의 없었다. life의 경우 그 반대 상황인데, 이유는 그림 2에서 보듯이 메모리 최고점 부분에서만 재사용되었다.

## 5 메모리 수거 및 실행 시간에 미치는 효과

실험 대상 프로그램들의 실행 시간 중 GC가 차지하는 비중은 1.1%–81.3%이었고 변환에 의해 GC 시간이 −1.6%–88.5% 단축되었다. GC 시간 단축 정도는 메모리 재사용률과 상관관계가 있다. 즉, 메모리 재사용률이 높으면 GC 시간도 크게 단축되었다. 그림 3의 (a)의 $c$ 열이 GC 시간의 변화를, 그래프 (b)가 상관 관계를 보여준다.

메모리 재사용에 따른 실행 시간의 변화는 여러가지 요소가 작용하였다. 그림 3.(a)의 $d$ 열, 그리고 그래프 (c)에 실험

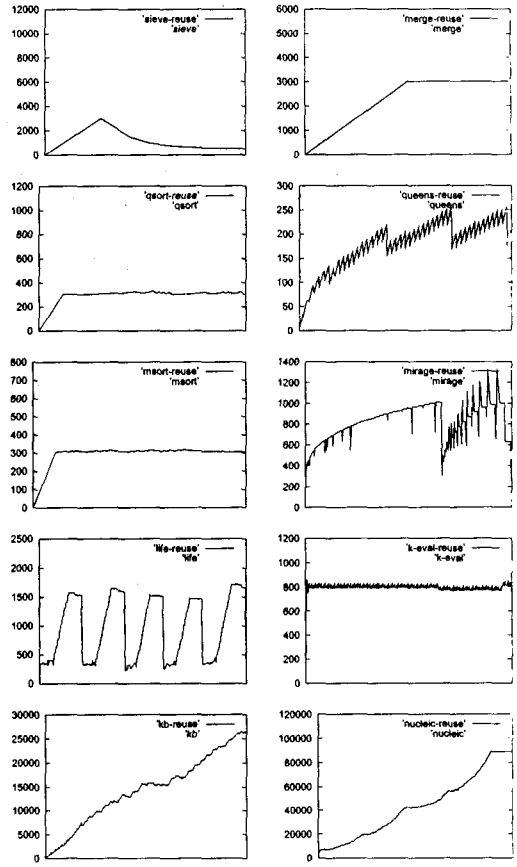



그림 2: 프로그램의 실행 시작부터 끝까지 살아 있는 메모리 셀의 수. 위쪽 점선은 원래 프로그램의 것이고 아래 실선은 변환된 프로그램의 것이다.

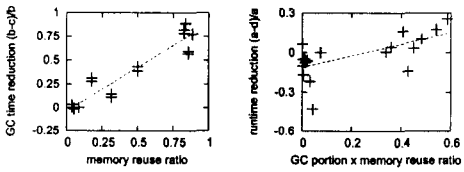
결과가 있다. 실행 속도가 25.4% 단축된 것부터 42.9% 늦어진 것까지 있다. qsort의 경우 메모리 재사용 비율과 실행 시간의 GC 비중이 모두 높아 실행 시간이 25.4% 단축되었다. mirage의 경우 재사용 비율은 높지만 GC 비중이 거의 없어 실행 시간이 42.9% 늦어졌다.

메모리 재사용은 GC 시간을 줄여 다섯 개의 프로그램에 대해 속도를 단축시켰다. merge, qsort, msort는 3.4%–25.4% 정도 속도가 단축되었는데, 비중이 큰 GC 시간을 많은 재사용을 통해 단축시켰기 때문이다. nucleic과 sieve의 경우도 유사하나 그 정도가 작다. 일부는 변환시 추가하는 논리값 인자를 처리하기 위한 부담때문에 느려지기도 했다.

본 실험을 위한 구현에서 메모리 재사용은 메모리 생성보다 비싸게 구현되었다. Objective Caml 3.06 (이하 OCaml) 컴파일러를 기반으로 구현하였는데, 메모리 셀을 생성할 때 무조건 수정 가능한 셀이라 초기화시키고, 메모리 재사용 명령어를 OCaml의 메모리 내용 수정 연산자 <-로 컴파일했다. OCaml 컴파일러에서 메모리 내용 수정은 메모리 생성보다 비싸게 구현되어 있다. OCaml은 두 세대 메모리 재활용 시스템(two generational garbage collection)을 사용하고 있는데,

| 프로그램 | $a$ 실행 | $b$ | $b/a$ GC | $c$ | $(b-c)/b$ 재사용시 GC | $d$ | $(a-d)/a$ 재사용시실행 |
|---|---|---|---|---|---|---|---|
| sieve | 0.78 | 0.388 | 49.7% | 0.159 | 59.0% | 0.89 | -14.1% |
| merge | 0.24 | 0.172 | 71.7% | 0.106 | 38.4% | 0.23 | 4.2% |
| qsort | 0.67 | 0.468 | 69.9% | 0.054 | 88.5% | 0.50 | 25.4% |
| queens | 0.33 | 0.126 | 38.2% | 0.128 | -1.6% | 0.35 | -6.1% |
| msort | 0.39 | 0.212 | 54.4% | 0.048 | 77.4% | 0.35 | 10.3% |
| mirage | 0.21 | 0.011 | 5.2% | 0.002 | 81.8% | 0.30 | -42.9% |
| life | 0.49 | 0.008 | 1.6% | 0.008 | 0.0% | 0.54 | -10.2% |
| k-eval | 0.41 | 0.007 | 1.7% | 0.006 | 14.3% | 0.48 | -17.1% |
| kb | 0.40 | 0.122 | 30.5% | 0.118 | 3.3% | 0.43 | -7.5% |
| nucleic | 0.45 | 0.187 | 41.6% | 0.135 | 27.8% | 0.45 | 0.0% |
| sieve | 7.15 | 2.817 | 39.4% | 1.224 | 56.5% | 7.11 | 0.6% |
| merge | 1.69 | 1.374 | 81.3% | 0.777 | 43.4% | 1.42 | 16.0% |
| qsort | 4.97 | 3.205 | 64.5% | 0.385 | 88.0% | 4.10 | 17.5% |
| queens | 2.82 | 0.950 | 33.7% | 0.954 | -0.4% | 3.01 | -6.7% |
| msort | 2.96 | 1.501 | 50.7% | 0.346 | 76.9% | 2.86 | 3.4% |
| mirage | 1.85 | 0.073 | 3.9% | 0.016 | 78.1% | 2.25 | -21.6% |
| life | 5.19 | 0.057 | 1.1% | 0.057 | 0.0% | 4.86 | 6.4% |
| k-eval | 3.78 | 0.044 | 1.2% | 0.039 | 11.4% | 3.97 | -5.3% |
| kb | 2.85 | 0.727 | 25.5% | 0.727 | 0.0% | 2.99 | -4.9% |
| nucleic | 2.70 | 1.158 | 42.9% | 0.797 | 31.2% | 2.70 | 0.0% |

(a) 실행 시간의 변화. Objective Caml 3.06 [4] 컴파일러로 컴파일하였고, 위쪽은 Intel P4 3.0C, Linux RedHat 9.0, 아래쪽은 Sun UltraSparc 400Mhz, Solaris 2.7에서 실행시켰다.




(b) GC 시간과 재사용률. (c) 실행시간과 재사용률×GC비중.

$a$ 초: 변환 전 프로그램의 실행 시간.
$b$ 초: 변환 전 프로그램의 메모리 수거 시간.
$c$ 초: 변환된 프로그램의 메모리 수거 시간.
$d$ 초: 변환된 프로그램의 실행 시간.

그림 3: 메모리 재사용 변환에 따른 실행 시간 변화.

젊은 세대(young generation)에서 메모리 수거를 할 때, 늙은 세대(old generation)로부터의 포인터(pointer)를 고려하여야만 한다. 즉, 셀의 내용을 수정할 때, 셀이 어느 세대에 있는지, 수정 전 값이 어느 세대를 가리키고 있는지, 수정할 값이 어느 세대를 가리키고 있는지 확인해야 한다. 수정 대상 셀이 늙은 세대에 있고 수정할 포인터가 젊은 세대를 가리키면 별도의 참조 테이블을 수정해 주어야 한다. 게다가 OCaml은 늙은 세대의 메모리 수거를 위해 표시 후 수거 (mark-and-sweep) 방법[3]을 쓰고 있는데, 셀의 내용을 수정할 때 이를 위한 표식도 수정해 주어야 한다. 현재의 구현이 최적은 아니지만 일반적으로 다세대 메모리 재활용 시스템(generational garbage collection)[3]에서 메모리 수정을 생성보다 싸게 구현하기는 어려울 듯 하다.

mirage가 느려진 것은 이런 비싼 메모리 재사용 때문이다. 메모리 재사용 비율은 높으나 GC 부분은 0에 가깝다. 그러므로 실행 시간에 영향을 주는 것은 비싼 재사용의 부담뿐이다.† 같은 이유로 k-eval과 life도 느려졌으나, 재사용률이 높지 않아 정도가 덜하다.

†추가의 논리값 인자를 처리하는 부담은 변환 전 프로그램의 실행 시간의 8.1% 미만이었다.

## 6 결론

실험 결과를 요약하면 다음과 같다.

- 분석 및 변환 비용은 비싸지 않다. Pentium4 3.0C에서 초당 1,582줄–29,000 줄이다.
- 총 생성된 메모리의 3.8%–88.6%를 재사용했다. 비율이 낮은 경우는 프로그램이 수행 중에 그래프 형태의 공유된 자료 구조를 많이 사용해서이다.
- 메모리 최고점을 0.0%–71.9% 하락시켰다. 하락이 낮은 경우는 메모리 재사용 비율이 낮거나, 최고점에 해당하는 셀들을 재사용 못해서이다.
- 메모리 수거 시간을 -1.6%–88.0% 단축시켰다. 증가된 경우는 메모리 수거 지점이 변경되면서 원래 지점보다 살아있는 셀이 많은 지점으로 이동해서이다.
- 실행시간은 42.9% 느려진 것부터 25.4% 빨라 진것까지 있다. 실행 시간 단축은 메모리 수거 시간 단축에 의해서이다. 느려진 경우는 다세대 메모리 재활용 시스템에서 메모리 재사용이 비싸기 때문이다.

실험 결과를 통해 다음 상관 관계가 있음을 알게 되었다.

- 그림 1의 그래프 (d): 변환에 의해 많이 재사용되면 메모리 최고점이 크게 하락한다. 예외적으로 메모리 최고점과 관계없는 메모리 셀을 재사용하여 메모리 감소에 영향을 주지 않은 경우도 있다.
- 그림 3의 그래프 (b): 변환에 의해 많이 재사용되면 메모리 수거 시간이 많이 단축된다.
- 그림 3의 그래프 (c): 변환에 의해 많이 재사용되고 전체 실행 시간 중 메모리 수거가 차지하는 비중이 크면 실행 시간이 많이 단축된다.

### 참고 자료

[1] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML (Revised)*. MIT Press, 1997.

[2] Oukseh Lee, Hongseok Yang, and Kwangkeun Yi. Inserting safe memory reuse commands into ML-like programs. In *Proceedings of the Annual International Static Analysis Symposium*, volume 2694 of *Lecture Notes in Computer Science*, pages 171–188. Springer-Verlag, June 2003.

[3] Richard Jones and Rafael Lins. *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*. John Wiley & Sons, 1996.

[4] Xavier Leroy, Damien Doligez, Jacques Garrigue, Didier Rémy, and Jérôme Vouillon. The Objective Caml system release 3.06. Institut National de Recherche en Informatique et en Automatique, August 2002. `http://caml.inria.fr`.