論文 90-27-8-8

RISC 컴파일러의 레지스터 할당부 설계

(The Design of A Register Allocation Phase for RISC Compilers)

朴 鍾 得* 林 寅 七*

(Jong Deuk Park and In Chil Lim)

要 約

본 논문에서는 RISC 컴파일러 시스템 구현에 필요한 레지스터 할당부의 설계방식을 제안하고 실현한다. 제안된 레지스터 할당부는 C프로그램을 기계 독립적인 중간 코드로 컴파일하여 프로그램의 각 변수를 심볼릭 레지스터로 변환한다. 심볼릭 레지스터에 대하여 국소적 할당 과정을 수행한 후 미할당된 변수에 대해서는 그래프 컬러링 알고리듬을 적용하여 광역적 레지스터 할당을 수행한다. 제안된 레지스터 할당부는 RISC와 같은 대규모의 레지스터 화일(register file)을 가진 시스템을 대상으로 설계한다.

Abstract

This paper describes and implements a design method of register allocation as a required module of RISC compiler systems. It compiles a C program to a machine-independent intermediate language, translates each variable into symbolic register. After local allocation process for the symbolic registers, global register allocation is executed by applying the graph coloring algorithm. This register allocation phase is designed for a system with the large register file like RISC machines.

I. 서 론

오늘날 컴퓨터 시스템의 보다 빠른 수행 성능을 얻기 위해 RISC 아키텍춰 방식의 프로세서 개발이 활발히 이루어지고 있다. RISC는 명령어의 형식을 단일화하여 디코드 시간을 단축시킴으로써 가능한 가장 빠른 머신 사이클을 얻을 수 있고 칩면적을 대폭 줄일 수 있어서 대규모 레지스터 화일의 실현이 가능하여 프로세서와 메모리 간의 빈번한 데이타 이동에 따른 속도 저하를 방지할 수 있다. 따라서 RISC 컴파일러 설계자는 레지스터 화일의 효율적인 관리에 주안점을 두게 된다.

레지스터 할당은 오래전부터 그래프 컬러링 문제로 간주되어 왔다.[1,2,3] 그래프 상에 나타나는 노드는 프로그램의 변수를 나타내며 두 노드가 서로 같은 레지스터에 놓여질 수 없다면 하나의 에지(edge)로 연결된다. 그래프 컬러링 알고리듬은 그래프의 각 노드에 색깔을 할당하는 것으로 두 노드가 에지로 연결되어 있는 경우 다른 색깔을 부여해야 한다. 그래프를 색칠할 때 사용되는 색깔의 수는 레지스터 할당을 위한 가용 레지스터(available register)의 수이다. 레지스터 할당의 목적은 프로그램 변수를 레

*正會員, 漢陽大學校 電子工學科
(Dept. of Elec. Eng., Hanyang Univ.)
接受日字 : 1990年 5月 14日

지스터에 할당함으로써 실행시간을 최소화 시키는 것이다. 레지스터 할당은 국소적 할당과 광역적 할당으로 나눌 수 있다. 국소적 레지스터 할당은 프로그램을 기본 블럭으로 분할하였을 때 각 기본 블럭 범위 내에서 변수를 레지스터에 할당하는 방법이다. 반면에 광역적 레지스터 할당에서는 색칠될 변수 결정에 있어서 전체 프로시쥬어를 고려한다.[2,5,6,10]

그래프가 한정된 색깔로 칠하여질 수 있는지를 결정하는 표준 컬러링 알고리듬은 NP완전(NP-complete)하다. 이 알고리듬은 전체 노드의 색깔을 결정한 후 색깔의 수가 가용 레지스터의 갯수보다 많을 경우는 역추적(backtracking)하여야 한다. 이 알고리듬의 복잡도는 첫 시도가 성공했을 때만 선형시간으로 된다. 그러나 주어진 그래프가 가용 레지스터의 수와 동일하거나 더 적은 수의 색깔로 색칠되지 않는다면 역추적하여 모든 가능한 색칠을 시도할 필요가 있으므로 지수증가적인 복잡도를 갖는다. 그러므로 표준 컬러링 알고리듬은 대상 머신(target machine)이 RISC와 같이 다수의 레지스터를 가질 때만 잘 적용된다. 즉 기존의 CISC 머신처럼 레지스터의 갯수가 한정되어 있는 경우는 최적화가 필요한 경우에만 표준 컬러링 알고리듬을 적용시켜, 컴파일시 레지스터 할당에 과다한 시간이 걸리지 않도록 한다.

레지스터 할당시에 변수의 사용 빈도를 측정하는 기법이 제안된 바 있는데 이 알고리듬은 현재 국소적 레지스터 할당의 기본 모델이 되고 있다. 70년대 초반에 그래프 컬러링 기법이 도입되었으며 IBM 에서는 실험적인 PL/I 컴파일러 설계에 이를 적용하여 레지스터 할당을 실제로 구현하였다. 또한 최근에는 노드에 우선 순위를 부여하는 광역적 할당 알고리듬이 제안된 바, 표준 컬러링 알고리듬이 보다 효율적인 코드 생성을 위해 본래의 이론적 기법에서 많은 진보가 이루어진 셈이다.[3,5,6]

그러나 Portable C Compiler(PCC)와 같은 상용컴파일러의 전반부에서는 레지스터 선언 변수에 대해서는 기존 레지스터에 우선적으로 할당되며 일반 변수는 프레임 포인터(frame pointer)를 이용하여 액세스(access)되므로 이런 실질적인 측면을 고려할 필요가 있다. 따라서 프레임 포인터에 따라 액세스되는 변수일지라도 가용 레지스터가 있다면 레지스터에 할당하는 것이 바람직하며 각 변수가 최대한 작은 수의 레지스터로 할당되도록 하는 효율적인 알고리듬이 요구되고 있다. 본 논문에서는 컴파일러 설계시에 적용할 수 있는 실제적인 레지스터 할당 기법을 제안하여 RISC 컴파일러 실현에 도움을 줄 수 있도록 한다. 이 기법은 예비처리(preprocessing) 과정 및 국소적 할당과 광역적 레지스터 할당으로 분할 구성된다. 즉 예비처리 과정에서는 변수를 심볼릭 레지스터로 변환하며, 국소적 할당에서는 데이타 흐름 분석에서 얻어지는 각 변수의 live range가 기본 블럭 내부에만 존재하는 변수에 대하여 색칠한다. 광역적 알고리듬은 국소적 할당에서 색칠되지 못하는, live range가 기본 블럭의 범주를 벗어나는 변수에 대하여 적용한다.

본 논문에서는 C프로그램의 컴파일시 RISC 아키텍춰에 대하여 효율적인 코드 생성을 할 수 있도록 제안된 알고리듬을 SPARC 워크스테이션 상에서 C 언어로 실현한다.

Ⅱ. 레지스터 할당부의 설계배경

컴퓨터들은 제각기 다른 형태의 기억장소를 가진다. 일반적으로, 기억장소는 메모리(memory)와 캐쉬 메모리(cache memory), 그리고 레지스터(register)의 세가지로 분류할 수가 있다. 이 중에서 컴퓨터의 수행속도에 가장 많은 영향을 미치는 요소는 레지스터로서 CPU와 동일 칩 상에 설계되어 있는데, RISC를 근간으로 하는 컴퓨터에서의 명령어들은 대부분 이러한 레지스터의 사용을 위주로하여 만들어진다. 프로그램의 실행시 프로세서는 데이타를 저장해야할 기억장소가 필요하다. 이를 위해 CPU는 내부에 고속의 기억장소인 레지스터 집합을 갖게 된다.

1. 레지스터 관리의 필요성

레지스터를 피연산자로 갖는 명령어는 메모리를 피연산자로 갖는 명령어 보다 더욱 빠르게 수행된다. 따라서 레지스터를 효과적으로 이용하는 것은 좋은 오브젝트 코드를 생성하는데 중요한 요소가 된다. 특히 최근의 동향이 많은 레지스터를 갖고 있는 RISC에 기울어지면서 그 중요성은 더욱 가중되고 있다.

레지스터 할당에 일반적으로 쓰이고 있는 그래프 컬러링 이론은 그래프 상에서 에지로 연결된 두 노드는 서로 다른 색깔을 가져야 한다는 것이다. 만일 그래프에서 사용할 수 있는 색깔의 수가 n이라면 그 그래프에 대한 그래프 컬러링은 n-컬러링이라고 한다.

레지스터 할당을 수행할 때 그래프상에 나타나는 노드들은 할당의 대상이 되는 변수들이고 노드들간에 연결되는 에지는 같은 레지스터를 공유할 수 없음을 의미하며, n은 가용 레지스터(available register)의 갯수이다. 사용할 수 있는 색깔의 수 n이 2보다 큰 자연수일 경우 어떤 그래프 G가 n종류의 색깔로 칠하여 질 수 있는지를 결정하는 문제는 NP 완전하

다. 말하자면, 이러한 결정은 첫 시도가 성공하였을 때에만 선형시간으로 나타나고 그렇지 않을 경우에는 역 추적하여 모든 가능한 색칠을 시도할 필요가 있으므로, 그래프 G의 사이즈(일반적으로 그래프의 사이즈는 그래프상의 노드와 에지수에 의존한다.)에 대해 지수 증가적인 시간으로 나타난다. 그러나 레지스터 할당에 관한 많은 연구에 의하면 그래프 컬러링의 NP완전 문제는 레지스터 할당을 하는데 있어서 그다지 큰 장애가 되는것은 아니라는 사실이 실험적으로 증명이 되고 있는데, 이는 이론상에서 도입되는 그래프는 모든 극단적인 경우를 모두 고려하는데 반하여 이론을 적용해서 레지스터 할당을 설계할 때 실제로 만들어 지는 그래프에 대해서는 그러한 극단적인 경우가 일어나는 일은 드물다. 표준 컬러링 알고리듬은 가용 레지스터의 갯수가 부족한 상황을 고려하여 변수들을 메모리로 대피(spill) 시키는 문제를 포함한다. 컴퓨터가 많은 수의 레지스터를 갖고 있다 하더라도 그 갯수에는 제한이 되어 있기 때문에 프로그램에 따라서는 레지스터에 상주해야 할 데이타의 갯수가 컴퓨터가 갖고있는 레지스터의 갯수를 초과하는 경우가 발생하기도 한다. 이때에는 레지스터내의 특정 데이타를 메모리에 대피시켜 다른 데이타가 레지스터에 상주할 수 있도록 해야한다. 결과적으로 그 데이타에 대해 메모리로의 store 명령이 필요하게 되고 뒷 부분에서의 창조를 위한 load명령이 뒤따르게 된다.

상대적으로 수행속도가 빠른 레지스터를 효율적으로 이용하는 레지스터 할당 알고리듬을 설계하여 가급적이면 불필요한 load/store 명령을 줄임으로써 컴파일러의 수행 속도를 향상시키는 문제는 최적화 컴파일러의 설계에 있어서 대단히 중요한 요소가 된다.

2. 레지스터 화일의 이용

RISC 아키텍춰를 크게 두가지로 분류해 보면 레지스터 윈도우(register window)를 사용하는 berkeley 형태와 캐쉬 메모리를 중점적으로 사용하는 stanford 형태로 나뉘게 된다. Berkeley 형태의 대표적인 예는 SPARC 프로세서[9]를 들 수가 있고 stanford 형태의 대표적인 예는 MIPS 프로세서[10]를 들 수가 있다.

본 연구에서는 레지스터 윈도우를 사용하는 컴퓨터를 근간으로 해서 이식성이 용이한 레지스터 할당 알고리듬을 설계되므로 레지스터 윈도우의 특징을 간단히 살펴보기로 한다.

대부분의 프로그램에서는 프로시쥬어 호출(procedure call)이 빈번히 발생하는데, 프로시쥬어 호출이 발생하게 되면 인수(argument)들이 호출된 프로시쥬어에 전달되고 국소 변수들(local variable)은 레지스터에서 메모리로 저장되어야 한다. 그렇게 함으로써 레지스터들은 호출된 프로시쥬어에 의해 사용될 수 있는 상태가 된다. 반대로 호출 프로시쥬어로의 복귀(return)가 발생하면, 메모리에 저장되었던 호출 프로시쥬어의 국소 변수들이 레지스터로 다시 load되고 현재 프로시쥬어의 계산 결과가 호출 프로그램으로 넘겨져야 한다. 이렇게 되면 load 및 store가 많이 발생하게 되어 실행시간 증가의 요인이 될 수 있다. 여기에 대한 해결책으로 나타난 것이 다수의 레지스터 집합을 이용하여 레지스터 윈도우를 형성하는 것이다.

레지스터 윈도우의 기본개념은 그림 1에 나타난 바와 같이, 프로시쥬어의 호출이 발생하면 자동적으로 CPU로 하여금 고정된 사이즈의 다른 레지스터들을 이용하게 함으로써 레지스터의 내용을 메모리로 store 하는 것을 대체하게 하므로 프로그램의 실행시간을 감소시킬 수 있다. 한편 이웃하는 프로시쥬어들 간에는 윈도우가 겹치게 되는데(overlapping)이는 인수들의 이동을 허용해주기 위함이다.

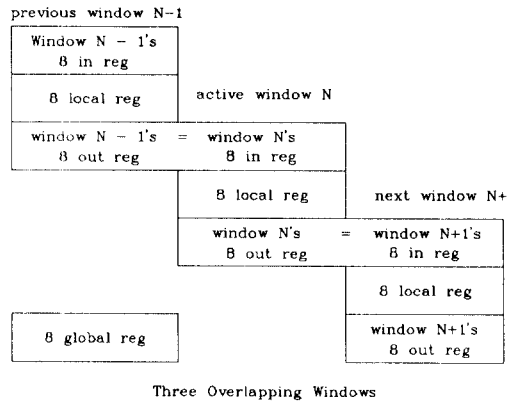

그림 1. 겹침 레지스터의 윈도우와 global 레지스터들
Fig. 1. Overlapping register windows and global registers.

그림 1에 나타난 바와 같이 각 윈도우는 3개의 고정 영역으로 나뉘어진다. in 레지스터들은 호출 프로시쥬어로부터 넘겨받은 인수들을 갖고 있고 계산결과를 다시 넘겨준다. local 레지스터들은 국소 변수(local variable)를 위해 사용되는데 이는 컴파일러에 의해 지정이 된다. out 레지스터들은 현 프로시쥬어

의 인수들과 계산결과를, 호출한 프로시쥬어와 교환하기 위해서 사용이 된다. 한 프로시쥬어에서의 out 레지스터들은 사실상 그 프로시쥬어를 호출한 프로시쥬어에서의 in 레지스터들과 같은 레지스터들이 된다. 이렇게 레지스터들이 겹침으로 인해서 실질적인 인수들의 이동이 없이 프로시쥬어 호출이 가능하다.

Ⅲ. 중간 코드

컴파일러의 설계에 있어서 중간 코드가 차지하는 비중은 매우 높다. 컴파일러를 크게 전반부(front-end)와 후반부(back-end)로 나누었을 때, 전반부는 원시 언어(source program)를 중간 코드로 번역하고, 후반부는 번역된 중간 코드로부터 대상 코드(target code)를 생성한다. 중간 코드를 사용하지 않아도 원시언어가 직접 대상 코드로 번역이 될 수가 있으나, 기계 독립적인 중간 코드(machine-independent intermediate code)를 사용하게 되면, 다음과 같은 잇점이 따르게 된다.

○재대상화(retargetting)의 용이 : 새로운 기계에 대한 컴파일러를 생성함에 있어서, 기존 컴파일러의 전반부는 변경하지않고, 새로운 기계의 후반부만을 기술함으로써 새로운 컴파일러의 생성을 용이하게 한다.

○기계 독립적인 코드 최적화(machine-independent code optimization)의 적용 : 원시 언어는 일반적으로 소스 코드 레벨에서 액세스할 수 없는 상세한 동작을 내포하고 있으므로, 더 낮은 레벨로 확장된 경우에 최적화 될 가능성이 많다. 일반적으로 레벨이 낮으면 낮을수록 수행되는 최적화를 발견할 수 있는 기회는 더욱 많게 되나 레벨이 너무 낮으면, 기계자체의 특성에 많이 의존하게 되어 이식성(portability)이 용이하지 않게 되므로 적합하지 않다. 중간 코드 상에서의 기계 독립적인 최적화 수행은 이러한 문제점들을 해결하여 준다.

3주소 코드는 명시된 이름이 그래프상의 내부노드에 해당하는 구문트리이거나 DAG의 선형화된 표현이다. 본 연구에서는 기계 독립적인 최적화가 수행되고 난 후의 3주소 코드 상에서 레지스터 할당알고리듬을 적용한다. 레지스터 할당을 위해 입력으로 들어오는 텍스트는 3주소 코드(3-address code)로 가정한다. 여기서 3주소 코드는 컴파일러의 파싱부에서 출력된 파싱 트리를 다시 기계 독립적인 중간 코드로 변환한 것이다. 3주소 코드의 일반형식은 x:=y op z이다. 여기서 x, y, z는 컴파일러에 의하여 생성된 임시 변수 또는 상수이고 op는 산술 연산자나 논리 연산자이다. 다음은 3주소 코드의 형식을 나타낸 것이다.[7][8]

○x:=y op z
○x:=op y
○x:=y
○goto L
○if x relop y goto L

조건 분기문으로써, x와 y에 대해 관계 연산자(<, <=, ==, !=, >=, >등)를 적용하여 x가 y에 대해 relop을 만족하면, 레이블 L이 있는 문으로 실행을 옮기고 그렇지 않은 경우에는 "if x relop y goto L" 다음의 문을 실행하게 된다.

○x:=&y, x:=*y
x:=&y에서 x의 값은 y의 주소가 되고,
x:=*y에서 x의 값은 y의 값이 나타내는 주소의 내용이 된다.

다음은 간단한 C프로그램에 대해 3주소 코드를 나타낸 것이다.

```
main( )
{
  int i, s;
  s=0;
  for(i=1;i<=20;i++)
    if(i<8 || i>11)
      s=s+i;
}
```

```
 1  main:
 2    t1 := =fp-8
 3    *t1 := =0
 4    t2 := =fp-4
 5    *t2 := =1
 6    L18 :
 7    t3 := =fp-4
 8    t4 := =*t3
 9    t5 := =t4<=20
10    if ! t5 goto L17
11    t6 := =fp-4
12    t7 := =*t6
13    t8 := =t7<8
14    t9 := =fp-4
15    t10 := =*t9
16    t11 := =t10>11
17    t12 := =t8 || t11
18    if ! t12 goto L19
19    t13 := =fp-4
20    t14 := =fp-8
21    t15 := =*t14
22    t16 := =fp-4
```

```
23  t17 : = * t16
24  t18 : = t15+t17
25  * t13 : = t18
26  L19 :
27  L16 :
28  t19 : = fp-4
29  t20 : = * t19
30  * t19 : = t20+1
31  goto L18
32 L17 :
33  ret
```

위 3주소 코드 중 문번호 25와 26에 레이블이 이중으로 된 것은 파싱에 따른 중간코드 생성시 원래는 다른 레이블로 번호가 매겨졌으나 레이블의 위치가 동일하게 된 경우이다.

Ⅳ. 레지스터 할당 알고리듬

제안된 레지스터 할당부는 그래프 이론에 기초를 두고 국소적 레지스터 할당과 광역적 레지스터 할당으로 분할 적용하여 설계한다. 레지스터 할당부는 특히 RISC와 같이 대 용량의 레지스터를 갖는 머신에 적합하기 때문에 RISC 컴파일러를 설계하는데 있어 매우 중요한 부분이 된다.

레지스터 할당과정은 3주소 코드로 표현되는 중간코드에서 나타나는 임시 변수(temporary variable), 국소 변수(local variable), 광역 변수(global variable)들을 n개의 심볼릭 레지스터(symbolic register)에 할당하여, 사용하는 레지스터의 갯수를 최소화하고, n-컬러링(n개의 레지스터에 변수를 할당하는 것) 이 불가능하다는 것이 발견될 때 필요한 메모리의 대피 횟수를 최소화 시키는데 중점을 둔다.

제안된 레지스터 할당부는 전체 컴파일러의 구성도에서 볼 때 다음 그림 2와 같이 중간 코드 생성기와 코드 생성기 사이에 위치한다.

레지스터 할당을 수행하기 전에 프레임 포인터로 액세스되는 단일 변수를 심볼릭 레지스터에 매핑시킨 후 할당 알고리듬을 적용한다. 고급 언어로는 일반적으로 널리 이용되는 C를 사용하며 파서(parser)에서 출력된 파싱 트리를 중간 코드화하여 예비처리 과정을 통해 변수를 가능한 많은 심볼릭 레지스터로 바꾸고 나서 레지스터 할당 알고리듬을 수행한다. 레지스터 할당부의 구성도는 그림 2와 같다.

레지스터 할당은 다음과 같은 3단계 과정을 수행한다.

○광역적 데이타 흐름 분석 (global data flow analysis)

그림 2. 레지스터 할당부의 위치
Fig. 2. The location of register allocation phase.

○예비처리 및 국소적 레지스터 할당
○광역적 레지스터 할당
이에 대해 자세히 살펴보면 다음과 같다.

1. 광역적 데이타 흐름 분석

레지스터 할당을 수행하는데 있어서 첫번째 단계는 광역적 데이타 흐름분석을 통하여 기본블럭 설정, 흐름 그래프 구성, 루프 검출, live range 결정등을 수행하게 된다.

기본 블럭이란 일련의 연속적인 문들로서 제어의 흐름이 기본 블럭의 첫문으로만 들어오고 마지막 문에서만 나가게 되어 있는 특성을 갖고 있다. 즉, 기본 블럭내의 첫 문이 실행이 되면 그 블럭내의 문들은 차례대로 수행이 된다.

흐름 그래프는 기본 블럭을 노드로 하고, 제어의 흐름을 에지로 하는 G=(N, A, s)형식의 방향성 그래프로서, 여기서 N은 각 노드의 집합, A는 에지로 나타나는 아크(arc)의 집합이며, s는 초기 노드를 나타낸다. 이러한 흐름 그래프는 초기 노드 s로부터, 모든 노드로 적어도 하나의 경로가 존재하는 특성을 가진다. 다음은 기본 블럭으로 분할한 후에 흐름 그래프를 구성하는 알고리듬을 나타내고 있다.

```
flow_graph( )
{
  for(모든 기본 블럭에 대하여)
    if(block[i].last가 if문이면)
    /*block[i].last:i블럭의 마지막 문 번호*/
      bnext[i][1]:=i+1;/*bnext[i][ ]는 i블럭에서
        제어의 흐름 */
      bnext[i][2]:=l1;/*l1은 goto가 가리키는 레이블
        이 속한 블럭 번호 */
    else if(block[i].last가 jmp나 goto문이면)
      bnext[i][1]:=l2;/*l2는 jmp가 가리키는 레이블
        이 속한 블럭 번호 */
    else if(block[ ].first가 레이블이면)
      bnext[i-1][1]:=i;
    end if
  end for
}
```

2. 루프 검출

흐름 그래프 상에서 역방향 에지(backward edge)는 하나의 루프를 형성하게 되는데 광역적 레지스터 할당시에 루프내의 각 노드에 대한 frequency weight를 계산하여 그 frequency weight가 높은 노드부터 우선적으로 레지스터를 할당하도록 한다. 루프내의 노드에 대한 frequency weight는 루프내에서 존재하는 모든 순방향 경로(forward path)를 조사한 후, 노드의 각 순방향 경로에 대한 중첩도를 계산함으로써 구하게 된다.

예를들면, 루프L에 n개의 순방향 경로가 존재할 때 루프내의 노드 i에 대한 frequency weight fw(i)는 다음과 같이 구할 수가 있다.

fw(i)=k(k<=n)
;k는 i가 속한 루프 내에서 i가 그 루프내의 모든 경로에 중첩되는 횟수
fw(i)=0 ;i는 루프내에 있지 않은 노드

다음은 루프에 대한 frequency weight를 검출하는 알고리듬을 나타내고 있다.

```
loop-detection( )
{
  for(모든 블럭에 대해)
    if(bnext[i][ ]<i)/*역방향 에지*/
      for(bnext[i][ ]부터 i까지 모든 블럭에 대해)
        freq_weight(k);
          /*loop내의 노드에 대한 frequency weight를
          계산, k는 loop내의 노드*/
      end for
    end if
  end for
}
freq_weight(k)
{
  i:=0;
  for(k가 속한 루프내의 모든 경로 Pn에 대해)
    if(k가 경로 Pn에 속하면)
      fw(k):=i+1;
    end if
  end for
}
```

3. Live range 결정

각 변수에 대한 live range를 결정하는데 있어서 선행되어야 할일은 각 변수에 대한 define과 use를 나타내는 def-use 체인 정보를 구하는 것이다. x:=y+z형태의 3주소 코드에서 x는 define(혹은 write)되었다고 하고 y와 z는 use(혹은 read)되었다고 한다. 이렇게 하여 구한 def-use 체인 정보를 통하여, 어떤 변수가 한 지점에서 live하다는 것은 그 변수가 그 지점 이후에서 use되는 경우를 말한다. live range는 초기 노드를 만날때까지 역방향(backward)으로 검색이 되며, 상충 그래프 구성을 위한 중요한 요소가 된다.

4. 예비처리 및 국소적 레지스터 할당

위 예제의 3주소 코드에 대해서 예비처리 과정이 우선적으로 수행이 되는데, 예비처리 과정이란 메모리에 할당이 되어 프레임 포인터(frame pointer)를 통해 액세스되는 변수중 단일 변수를 심볼릭 레지스터에 할당하는 것이다. 국소적 레지스터 할당은 임시변수와 live range가 기본 블럭 내에 국한되는 변수에 대해 적용된다. 그렇기 때문에 그 대상 변수에 할당된 레지스터는 다음 블럭에서 다른 변수를 위해 재할당될 수 있다는 특징을 갖는다. 국소적 레지스터 할당을 위해 다음과 같이 변수에 대하여 W(write), L(live), R(read), D(dead)관계를 구한다.

○기본 블럭 내에서 대상 변수(주로 임시 변수)가 define된 문 번호상에 W를 놓는다.

○W 이후에 use된 지점의 문번호 상에 R을 놓는다.

○W와 R사이의 문 번호는 L을 놓는다.

○기본 블럭 내에서 W, R, L 이외의 문 번호는 공백으로 놓는다.

위 예제 프로그램에서 문번호 11-18의 기본 블럭에 대해서 W, R, L, D 행렬을 구하면 아래와 같다.

	6	7	8	9	10	11	12
11	W						
12	R	W					
13		R	W				
14			L	W			
15			L	R	W		
16			L		R	W	
17			R			R	W
18							R

(공란은 모두 D 상태가 된다.)

한 변수가 W되면 원칙적으로는 별도의 심볼릭 레지스터를 할당하지만 W가 있는 동안 문 번호 상에 다른 변수의 D가 있거나 R이 있더라도 R다음의 문 번호가 D가 되어있는 경우는 그런 변수들 중 맨처음 변수의 심볼릭 레지스터 부터 재차 할당함으로써 할당하는 심볼릭 레지스터의 갯수를 절약할 수가있다.

한편 본 연구에서는 3주소 코드로 나타내는 트리플 표현을 중간 코드로 사용하기 때문에 동일 문 번호 상에서 복수개의 W가 나타날 수 없다. 위의 행렬을 통하여 국소적 레지스터 할당을 적용시키면 변수6은 가용 레지스터 중 제일 처음 나오는 심볼릭 레지스터에 할당하고 변수 7 이 write될 때, 변수 6 이 먼저 read되고 변수 6 은 문 번호 13이후에는 dead 되므로 변수 7 은 변수 6 과 같은 심볼릭 레지스터에 할당할 수가 있다. 변수 8 의 경우도 변수 7 과 마찬가지로 동일한 심볼릭 레지스터에 할당이 되나, 변수 9 는 write시에 변수 8 이 live 상태에 있으므로 변수8과 같은 심볼릭 레지스터에 할당을 하지 못하고 그 다음으로 이용 가능한 심볼릭 레지스터에 할당을 한다.

다음은 예비처리 및 국소적 레지스터 할당 알고리듬을 나타내고 있다.

```
pre_loc_reg_allo( )
{
  for(fp를 통해 액세스되는 값이 초기화 되는 변수에 대해)
    k:=fp-1;/*i는 offset값*/
    r_k:=val[ ];/*k=1,2,3…, val[ ]:fp로 액세스되는 값*/
  end for
  for(fp를 통해 액세스되는 변수에 대해)
    for(모든 k에 대해)
      if(fp-i가 k와 같으면)
        변수를 r_k에 할당;
      end if
    end for
  end for/*예비처리 과정 끝*/
  for(모든 기본 블럭에 대해)
    for(모든 문에 대해)
      if(W만 있으면)/*블럭의 첫문*/
        r_p를 할당;
      else if(L이 없으면)/*L은 변수가 live한 상태*/
        r_p를 할당;
      else
        r_q를 할당/* rq는 live한 상태의 변수에
          할당된 레지스터를 제외한 레지스터*/
            /*p≠q*/
      end if
    end for
  end for
}
```

5. 광역적 레지스터 할당

광역적 레지스터 할당은 국소적 레지스터 할당을 수행한 변수들을 제외한 나머지 변수들에 대해서, 기

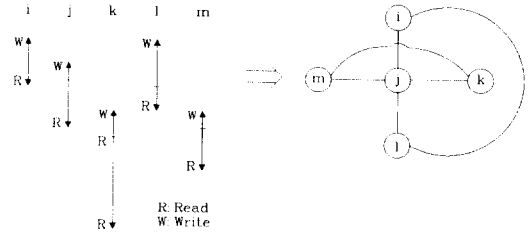

그림 3. 최대 잠재범위가 가장 긴 변수가 대피변수로 선택되지 않는 예

Fig. 3. An example of not chosen as a spill variable whose max dormant range is the longest of all.

본 블럭으로만 제한하지 않고 전체 프로그램을 통하여 수행한다. 광역적 레지스터 할당시 가용 레지스터의 갯수는 전체 가용 레지스터의 갯수에서 국소적 레지스터 할당시에 각 기본 블럭에 할당된 레지스터의 갯수를 뺀 나머지이다. 광역적 레지스터 할당은 대상 변수의 live 범위가 기본 블럭을 벗어나게 되는데, 다음과 같이 3단계를 거쳐서 할당을 한다.

○상충 그래프 구성
○대피 결정
○노드 컬러링

(1) 상충 그래프 구성

각 변수에 대한 live범위가 결정되면, 국소적 레지스터 할당을 수행하고 난 후 아직 할당되지 않은 변수들(live범위가 기본 블럭의 범주를 벗어나는 변수들)을 대상으로 상충 그래프를 구성한다.

상충 그래프 상에 나타내는 노드들은 광역적 레지스터 할당시의 대상 변수들이고 두 변수의 live 범위가 서로 겹치게 되면, 두 변수에 대응되는 노드들은 상충 그래프 상에서 에지로 연결된다. N개의 노드로 구성된 상충 그래프는 N*N비트 행렬로 표현되며, 노드가 인접되어 에지로 연결되면, 비트 행렬상의 i행과 j열의 M_{ij}비트는 1로 세트되고, 에지로 연결되지 않으면 0으로 세트된다. 그러나 두 변수의 live범위가 서로 겹친다 하더라도 겹치는 부분이 한 변수의 마지막 use이고 다른 한 변수의 define 지점이라면, 레지스터를 공유할 수가 있으므로 그 두 변수를 나타내는 노드들은 에지로 연결되지 않는다.

(2) 대피 결정

광역적 레지스터 할당 단계에서 상충 그래프를 구성하였을 때, 서로간에 에지로 연결된 두 노드는 같은 색깔을 가질수가 없다. (이는 두 변수가 하나의 레지스터를 공유할 수 없음을 의미)일반적으로 프로

그램의 양이 방대해지면 상충 그래프가 필요로 하는 색깔의 수는 가용 레지스터의 갯수를 초과하게 된다. (대피 조건). 대피조건이 발생하면 변수를 메모리로 대피시켜야 하는 경우가 발생하는데, 대피시에는 과도한 컴파일 시간이 요구되기 때문에 어느 변수를 대피 변수로 결정하느냐 하는 문제는 대피 횟수를 줄이고 대피 비용을 최소화하는데 있어서 중요한 요소가 된다.

대피변수의 결정은, 각 변수의 live 범위를 검색한 후 각 변수의 최대 잠재범위(define 된 후 마지막 use까지 실제로 use가 안되는 범위)및 상충 그래프 상에서의 에지수를 고려하여 결정한다. 일반적인 경우에는 최대 잠재범위가 가장 긴 변수가 불필요한 에지를 가장 많이 형성한다고 할 수 있으나 반드시 그렇지만은 않다. 그림 3에 나타난 바와 같이 최대 잠재범위가 가장긴 변수는 k인데 반하여, 상충 그래프 상에서 최대 에지수를 갖는 변수는 j이다. 하지만 에지수가 가장 많다고 해서 대피변수로 선택되는 것은 아니다. 즉, 에지수와 최대 잠재범위를 모두 고려해서 대피변수를 선택하게 된다. 말하자면 최대 잠재범위가 두번째나 세번째로 긴 변수(혹은 그 이상)가 대피변수로 먼저 결정이 될 수도 있다. 이렇게 하여 대피변수가 결정이 되면 대피변수의 live범위를 쪼개어 상충 그래프를 재 구성하고 대피시키는 부분에 store 명령어를, 후반부에서 참조되는 부분에 load 명령어를 삽입한다.

다음은 메모리 대피를 위한 알고리듬이다.

```
spill_var( )

  for(모든 광역 변수에 대해)
    s[ ]:=max_range[ ]+edge_no[ ];/*max_range[ ]:각
      변수의 최대 잠재범위, edge_no[ ]:각 변수의 에지 수*/
  end for
  s[ ]값이 가장 큰 변수를 spill변수로 선택;
  spill변수의 최대 잠재범위 시작 부분에 store명령 삽입;
  spill변수의 최대 잠재범위 끝 부분에 load명령 삽입;
```

만일 상충 그래프를 재 구성한 이후에도 대피조건이 발생한다면 또다른 대피 변수를 결정하여 대피조건이 발생하지 않을 때까지 알고리듬을 반복한다.

(3) 노드 컬러링

대피조건이 더이상 발생하지 않는 상충 그래프 상에서 수행한다. 상충 그래프 상의 노드를 따라 가용 레지스터 수 만큼의 색깔을 각 노드에 색칠한다. 에지로 연결된 두 노드는 동시에 live한 두 변수를 의미하므로 같은 색깔을 공유할 수가 없다.

다음은 각 노드에 색을 칠하여 주는 노드 컬러링 알고리듬을 나타내고 있다.

```
coloring( )

  for(spill변수를 제외한 모든 광역 변수에 대해)
    if(M_ij가 0이면)
      j번째 변수에 i번째 변수와 같은 레지스터를 할당;
      /*M_ij는 비트 행열에서 i행 j열 비트*/
    else if(M_ij가 1이면)
      j번째 변수에 i번째 변수와 다른 레지스터를 할당;
    end if
  end for
```

V. 실험 및 결과

대상 RISC 프로세서는 Fujitsu의 SPARC로 선정하였으며, SPARC는 겹침 윈도우 형식의 레지스터 화일을 가지고 있다. 제안된 레지스터 할당부는 SPARC가 탑재된 SUN-4머신 상에서 C언어로 구현되었다. 아래에 앞서의 예제 프로그램에 대한 레지스터 할당 후의 중간 코드를 나타내었다.

```
 1 main:
 2 r1 : =0
 3 r2 : =1
 4 L18 :
 5   t1 : =r2
 6   t1 : =t1<=20
 7   if  ! t1 goto L17
 8   t1 : =r2
 9   t1 : =r2<8
10   t2 : =r2>11
11   t1 : =t1 || t2
12   if  ! t1 goto L19
13   t1 : =r1+r2
14   r1 : =t1
15 L19 :
16 L16 :
17   r1 : =r1+1
18   goto L18
19 L17 :
20   ret
```

단, r1, r2는 광역적 레지스터 할당에 따라 적용된 심볼릭 레지스터이다. 표 1에 4개의 C벤치마크(bench mark)프로그램에 대하여 3주소 중간 코드를 생성

한 후 본 알고리듬을 적용시킨 결과를 나타낸다. 또한 본 알고리듬 수행 후 할당된 레지스터의 수를 SPARC 프로세서가 탑재된 SUN-4 워크스테이션에 있는 기존의 컴파일러가 할당한 레지스터의 갯수와 비교하고 있다. 표 1에서 감소 비율은 중간 코드 생성기에서 출력된 중간 코드의 수에 대하여 레지스터 할당 후의 코드와의 비율을 나타낸 것이다.

표 1. 레지스터 할당 결과
Table 1. The result of register allocation.

bench mark	레지스터 할당전 중간 코드와의 감소비율(%)	할당된 심볼릭 레지스터의 수	SUN-4에서 출력된 오브젝트 코드상의 레지스터의 수	SUN-4에서 최적화 수행후 할당된 레지스터의 수
bsearch	38	6	18	7
bubble	37	6	21	9
shaker	41.5	8	18	12
sieve	41.6	8	18	14

Ⅵ. 결 론

본 논문에서는 RISC 컴파일러 설계시 필수적인 레지스터 할당부의 효율적인 수행을 위하여 상용 RISC 머신에 적합한 알고리듬을 제안하고 실현하였다. 그리고 몇 가지의 C프로그램에 대하여 이를 적용한 결과를 측정하였다. 컴파일러의 전반부는 변수를 프레임 포인터를 통해 액세스하도록 하므로 실제의 레지스터에의 매핑 과정은 많은 어려움을 겪을수밖에 없다. 이러한 문제점을 해결하기 위하여 레지스터 할당을 수행하기전에 예비처리 과정을 두어 변수를 심볼릭 레지스터에 대입한 후 본 알고리듬을 수행하였다. 제안된 알고리듬을 SUN-4 머신에 탑재하여 기존의 SUN-4컴파일러의 최적화 및 레지스터 할당 수행 결과와 비교한 결과 더 많은 수의 레지스터를 줄일 수 있음을 입증하였다. 앞으로의 연구과제는 프로시쥬어 간의 레지스터 할당 알고리듬의 개발이다.

參 考 文 獻

[1] G.J. Chaitin, "Register allocation and spilling via graph coloring," ACM SIGPLAN Noitces, 17, 6 (June 1982). (Proceedings of the SIGPLAN 82 Symposium on Compiler Construction), pp. 201-207.

[2] F. Chow, "A portable machine-independent global Optimizer design and Measurements," *Ph. D. Thesis, Computer System* Lab, Stanford University, Dec. 1983.

[3] R.A. Freiburghouse, "Register allocation via usage counts," *Comm. ACM 17,11, Nov. 74.*

[4] B.W. Reverett, "Register allocation in optimizing compilers," Ph. D. Thesis, Carnegie-Mellon University, February 1981.

[5] F. Chow and J. Hennessy, "Register allocation by priority-based coloring," *Proceedings of the ACM SIGPLAN '84 Symposium on Compiler Construction SIGPLAN* Notices, vol. 19, Number 6, June 1984, pp. 222-232.

[6] G.J. Chaitin, et. al., "Register Allocation via Coloring," Computer Languages June 1981 pp. 47-57.

[7] A.V. Aho and R. Sethi and J.D. Ullman, Compilers Principles, Techniques, and Tools. Addison-Wesley, 1986.

[8] M.S. Hecht, Flow Analysis of Computer Programs. North-Holland, New York, 1977.

[9] M.G.H. Katevenis, "Reduce instruction set computer architectures for VLSI," University of California at Berkeley, Ph. D. Thesis, Oct. 1983.

[10] Paul Chow, The MIPS-X RISC Microprocessor. Kluwer Academic Publishers, 1989.

[11] 조정삼, 이형우, 이병노, 김주형, 황병현, 진은경, 박종득, 김은성, 임인칠, "RISC 머신을 위한 기계 독립적 optimizer의 설계," 대한전자공학회 하계종합학술대회 논문집, 제12권 제1호, pp. 304-307, 1989. 7.

[12] 이병노, 김주형, 황병현, 박종득, 김은성, 임인칠, "RISC 컴파일러 설계를 위한 레지스터 지정 및 할당 알고리듬," 한국정보과학회 가을학술발표논문집 vol. 16, no. 2, pp. 621-624, 1989.

[13] 박종득, 임인칠, "RISC 컴파일러의 기계 독립적 Global Optimizer 설계," 대한전자공학회 논문지, 제27권 제3호, pp. 368-374, 1990. 3.

著 者 紹 介

朴 鍾 得(正會員)
1959年 9月 13日生. 1983年 2月 한양대학교 전자공학과 졸업. 1985年 2月 한양대학교 대학원 전자공학과 졸업 공학석사 학위취득. 1985年 3月～현재 한양대학교 대학원 전자공학과 박사과정 재학중. 주관심분야는 RISC Compiler Design, Computer Architecture, Microprogramming 등임.

林 寅 七 (正會員) 第25卷 第8號 參照
현재 한양대학교 전자공학과 교수